LCOV - code coverage report

			Coverage	Total	H
Current view:	top level - build/src/plugins/intel_cpu/cross-compiled/SVE - executor_pa.cpp (source / functions)	**Lines:**	64.5 %	799	5
Test:	full_covergae.info	**Functions:**	44.1 %	68	
Test Date:	2025-04-10 19:26:12				

```
          Line data    Source code
       1            : // Copyright (C) 2018-2025 Intel Corporation
       2            : // SPDX-License-Identifier: Apache-2.0
       3            : //
       4            : #include <cfloat>
       5            : #include <cmath>
       6            : #include <cstring>
       7            : #include <iostream>
       8            : #include <limits>
       9            : #include <type_traits>
      10            :
      11            : #if defined(HAVE_AVX2) || defined(HAVE_AVX512F)
      12            : #    include <immintrin.h>
      13            : #endif
      14            :
      15            : #include "attn_memcpy.hpp"
      16            : #include "attn_quant.hpp"
      17            : #include "attn_quant_kernel.hpp"
      18            : #include "cache_rotation.hpp"
      19            : #include "common.hpp"
      20            : #include "executor_pa.hpp"
      21            : #include "executor_pa_common.hpp"
      22            : #include "openvino/core/parallel.hpp"
      23            : #include "openvino/core/type/bfloat16.hpp"
      24            : #include "openvino/core/type/float16.hpp"
      25            : #include "softmax_kernel.hpp"
      26            : #include "transpose_kernel.hpp"
      27            : #include "utils/plain_tensor.hpp"
      28            : #if defined(OPENVINO_ARCH_X86_64)
      29            : #    include "nodes/kernels/x64/brgemm_kernel.hpp"
      30            : #elif defined(OPENVINO_ARCH_ARM64) && defined(HAVE_SVE)
      31            : #    include "arm_sve.h"
      32            : #    include "nodes/kernels/aarch64/brgemm_kernel.hpp"
      33            : #    include "nodes/kernels/aarch64/sve_utils.hpp"
      34            : #    include "nodes/kernels/kai/kleidi_kernel.hpp"
      35            : #endif
      36            :
      37            : namespace ov::Extensions::Cpu::XARCH {
      38            :
      39            : using namespace ov;
      40            : using namespace ov::intel_cpu;
      41            :
      42            : // currently depends on brgemm which only support x64 or ARM SVE
      43            : #if defined(OPENVINO_ARCH_X86_64) || (defined(OPENVINO_ARCH_ARM64) && defined(HAVE_SVE))
      44            :
      45            : #    if defined(HAVE_AVX2) || defined(HAVE_AVX512F)
      46            :
      47            : #        define prefetch_bytes(bytes, sel, advance, src) \
      48            :             {                                           \
      49            :                 auto* p = reinterpret_cast<char*>(src);  \
      50            :                 for (size_t i = 0; i < bytes; i += 64)   \
      51            :                     _mm_prefetch(p + i + advance, sel);  \
      52            :             }
      53            :
      54            : #    else
      55            :
      56            : #        define prefetch_bytes(bytes, sel, advance, src)
      57            :
      58            : #    endif
      59            :
      60            : template <typename TA, typename TB>
      61        71059 : void cvt_copy(TA* dst, TB* src, size_t n) {
      62        71059 :     size_t i = 0;
      63            : #    if defined(HAVE_AVX512F)
      64            :     for (; i + vec_len_f32_avx512 <= n; i += vec_len_f32_avx512) {
      65            :         auto vb = mm512_uni_loadu_ps(src + i);
      66            :         mm512_uni_storeu_ps(dst + i, vb);
      67            :     }
      68            : #    elif defined(HAVE_AVX2)
      69            :     for (; i + vec_len_f32_avx2 <= n; i += vec_len_f32_avx2) {
      70            :         auto vb = mm256_uni_loadu_ps(src + i);
      71            :         mm256_uni_storeu_ps(dst + i, vb);
      72            :     }
      73            : #    elif defined(HAVE_SVE)
      74            :     if constexpr (std::is_same<TA, TB>::value) {
      75            :         auto pg_dst = sve_predicate<sizeof(TA)>();
      76            :         auto vlen = sve_vlen<sizeof(TA)>();
      77            :         for (; i + vlen <= n; i += vlen) {
      78            :             auto vb = svld1(pg_dst, src + i);
      79            :             svst1(pg_dst, dst + i, vb);
      80            :         }
      81            :         auto pgt = sve_predicate<TA, sizeof(TA)>(i, n);
      82            :         auto vb = svld1(pg_dst, src + i);
      83            :         svst1(pg_dst, dst + i, vb);
      84            :         return;
      85            :     } else if constexpr (std::is_same<TA, float>::value && std::is_same<TB, ov::float16>::value) {
      86        71059 :         auto src_ptr = reinterpret_cast<float16_t*>(src);
      87        71059 :         auto pg_vl2 = svwhilelt_b16(svcnth() / 2, svcnth());
      88        71059 :         auto vlen = svcnth() / 2;
      89        71059 :         auto pg_dst = svptrue_b32();
      90       355421 :         for (; i + vlen <= n; i += vlen) {
      91       272260 :             auto load_src = svld1_f16(pg_vl2, src_ptr + i);
      92       284362 :             auto src_interleave = svzip1_f16(load_src, load_src);
      93       284362 :             auto cvt_dst = svcvt_f32_f16_z(pg_dst, src_interleave);
```

```
      94  284362 :             svst1(pg_dst, dst + i, cvt_dst);
      95         :         }
      96         :     }
      97         : #    endif
      98   83161 :     for (; i < n; i++) {
      99     222 :         dst[i] = src[i];
     100         :     }
     101   82939 : }
     102         :
     103  263026 : size_t inline get_sub_byte_multiplier(ov::element::Type type) {
     104  263026 :     return one_of(type, ov::element::i4, ov::element::u4) ? 8 / type.bitwidth() : 1;
     105         : }
     106         :
     107         : template <typename T,
     108         :           ov::element::Type_t SRC_PREC,
     109         :           std::enable_if_t<(std::is_same_v<T, ov::bfloat16> || std::is_same_v<T, ov::float16> ||
     110         :                             std::is_same_v<T, float>)&&(SRC_PREC != ov::element::u8 || SRC_PREC != ov::element::u4),
     111         :                            bool> = true>
     112         : static void attn_acc_value_block(float* out,
     113         :                                   float* weight,
     114         :                                   T* v,
     115         :                                   const size_t S,
     116         :                                   const size_t block_size,
     117         :                                   const size_t group_size) {
     118         : #    if defined(HAVE_AVX512F)
     119         :     size_t j = 0;
     120         :     for (; j + 4 <= block_size; j += 4) {
     121         :         auto attn_w_vec0 = _mm512_set1_ps(weight[0]);
     122         :         auto attn_w_vec1 = _mm512_set1_ps(weight[1]);
     123         :         auto attn_w_vec2 = _mm512_set1_ps(weight[2]);
     124         :         auto attn_w_vec3 = _mm512_set1_ps(weight[3]);
     125         :         size_t i = 0;
     126         :         for (; i + vec_len_f32_avx512 <= S; i += vec_len_f32_avx512) {
     127         :             auto v_out = mm512_uni_loadu_ps(out + i);
     128         :             v_out = _mm512_fmadd_ps(attn_w_vec0, mm512_uni_loadu_ps(v + i), v_out);
     129         :             v_out = _mm512_fmadd_ps(attn_w_vec1, mm512_uni_loadu_ps(v + i + S), v_out);
     130         :             v_out = _mm512_fmadd_ps(attn_w_vec2, mm512_uni_loadu_ps(v + i + S * 2), v_out);
     131         :             v_out = _mm512_fmadd_ps(attn_w_vec3, mm512_uni_loadu_ps(v + i + S * 3), v_out);
     132         :
     133         :             _mm512_storeu_ps(out + i, v_out);
     134         :         }
     135         :         for (; i < S; i++) {
     136         :             out[i] += weight[0] * v[i];
     137         :             out[i] += weight[1] * v[i + S];
     138         :             out[i] += weight[2] * v[i + S * 2];
     139         :             out[i] += weight[3] * v[i + S * 3];
     140         :         }
     141         :         v += 4 * S;
     142         :         weight += 4;
     143         :     }
     144         :     if (j + 2 <= block_size) {
     145         :         auto attn_w_vec0 = _mm512_set1_ps(weight[0]);
     146         :         auto attn_w_vec1 = _mm512_set1_ps(weight[1]);
     147         :         size_t i = 0;
     148         :         for (; i + vec_len_f32_avx512 <= S; i += vec_len_f32_avx512) {
     149         :             auto v_out = mm512_uni_loadu_ps(out + i);
     150         :             v_out = _mm512_fmadd_ps(attn_w_vec0, mm512_uni_loadu_ps(v + i), v_out);
     151         :             v_out = _mm512_fmadd_ps(attn_w_vec1, mm512_uni_loadu_ps(v + i + S), v_out);
     152         :
     153         :             _mm512_storeu_ps(out + i, v_out);
     154         :         }
     155         :         for (; i < S; i++) {
     156         :             out[i] += weight[0] * v[i];
     157         :             out[i] += weight[1] * v[i + S];
     158         :         }
     159         :         v += 2 * S;
     160         :         weight += 2;
     161         :         j += 2;
     162         :     }
     163         :     if (j < block_size) {
     164         :         auto attn_w_vec0 = _mm512_set1_ps(weight[0]);
     165         :         size_t i = 0;
     166         :         for (; i + vec_len_f32_avx512 <= S; i += vec_len_f32_avx512) {
     167         :             auto v_out = mm512_uni_loadu_ps(out + i);
     168         :             v_out = _mm512_fmadd_ps(attn_w_vec0, mm512_uni_loadu_ps(v + i), v_out);
     169         :
     170         :             _mm512_storeu_ps(out + i, v_out);
     171         :         }
     172         :         for (; i < S; i++) {
     173         :             out[i] += weight[0] * v[i];
     174         :         }
     175         :     }
     176         :     return;
     177         : #    elif defined(HAVE_AVX2)
     178         :     size_t j = 0;
     179         :     for (; j + 4 <= block_size; j += 4) {
     180         :         auto attn_w_vec0 = _mm256_set1_ps(weight[0]);
     181         :         auto attn_w_vec1 = _mm256_set1_ps(weight[1]);
     182         :         auto attn_w_vec2 = _mm256_set1_ps(weight[2]);
     183         :         auto attn_w_vec3 = _mm256_set1_ps(weight[3]);
     184         :         size_t i = 0;
     185         :         for (; i + vec_len_f32_avx2 <= S; i += vec_len_f32_avx2) {
     186         :             auto v_out = mm256_uni_loadu_ps(out + i);
     187         :             v_out = _mm256_fmadd_ps(attn_w_vec0, mm256_uni_loadu_ps(v + i), v_out);
     188         :             v_out = _mm256_fmadd_ps(attn_w_vec1, mm256_uni_loadu_ps(v + i + S), v_out);
     189         :             v_out = _mm256_fmadd_ps(attn_w_vec2, mm256_uni_loadu_ps(v + i + S * 2), v_out);
     190         :             v_out = _mm256_fmadd_ps(attn_w_vec3, mm256_uni_loadu_ps(v + i + S * 3), v_out);
     191         :
     192         :             mm256_uni_storeu_ps(out + i, v_out);
     193         :         }
     194         :         for (; i < S; i++) {
     195         :             out[i] += weight[0] * v[i];
     196         :             out[i] += weight[1] * v[i + S];
     197         :             out[i] += weight[2] * v[i + S * 2];
```

```
                :             out[i] += weight[3] * v[i + S * 3];
                :         }
                :         v += 4 * S;
                :         weight += 4;
                :     }
                :     if (j + 2 <= block_size) {
                :         auto attn_w_vec0 = _mm256_set1_ps(weight[0]);
                :         auto attn_w_vec1 = _mm256_set1_ps(weight[1]);
                :         size_t i = 0;
                :         for (; i + vec_len_f32_avx2 <= S; i += vec_len_f32_avx2) {
                :             auto v_out = mm256_uni_loadu_ps(out + i);
                :             v_out = _mm256_fmadd_ps(attn_w_vec0, mm256_uni_loadu_ps(v + i), v_out);
                :             v_out = _mm256_fmadd_ps(attn_w_vec1, mm256_uni_loadu_ps(v + i + S), v_out);
                :
                :             mm256_uni_storeu_ps(out + i, v_out);
                :         }
                :         for (; i < S; i++) {
                :             out[i] += weight[0] * v[i];
                :             out[i] += weight[1] * v[i + S];
                :         }
                :         v += 2 * S;
                :         weight += 2;
                :         j += 2;
                :     }
                :     if (j < block_size) {
                :         auto attn_w_vec0 = _mm256_set1_ps(weight[0]);
                :         size_t i = 0;
                :         for (; i + vec_len_f32_avx2 <= S; i += vec_len_f32_avx2) {
                :             auto v_out = mm256_uni_loadu_ps(out + i);
                :             v_out = _mm256_fmadd_ps(attn_w_vec0, mm256_uni_loadu_ps(v + i), v_out);
                :
                :             mm256_uni_storeu_ps(out + i, v_out);
                :         }
                :         for (; i < S; i++) {
                :             out[i] += weight[0] * v[i];
                :         }
                :     }
                :     return;
                : #    endif
                :     for (size_t j = 0; j < block_size; j++) {
                :         for (size_t i = 0; i < S; i++) {
                :             out[i] += weight[j] * v[i];
                :         }
                :         v += S;
                :     }
                : }
                : template <typename T, ov::element::Type_t SRC_PREC, std::enable_if_t<SRC_PREC == ov::element::u8, bool> = true>
         136082 : static void attn_acc_value_block(float* out,
                :                                   float* weight,
                :                                   uint8_t* v,
                :                                   const size_t S,
                :                                   const size_t block_size,
                :                                   const size_t group_size) {
                :     // The layout for per token per head:
                :     // |scale(f32)|zeropoint(f32)|quantized feature(u8,idx_1)|quantized feature(u8,idx_2)|...|quantized
                :     // feature(u8,idx_S)| The quantized feature will start from 8bytes=sizeof(float)+sizeof(float)
         136082 :     size_t src_offset = 0;
         136082 :     size_t dst_offset = 0;
         136082 :     const size_t params_offset = sizeof(float) * 2;
         136082 :     const size_t src_stride = S / group_size * (group_size + params_offset);
                :
                : #    if defined(HAVE_AVX512F)
                :     size_t j = 0;
                :     for (; j + 4 <= block_size; j += 4) {
                :         dst_offset = 0;
                :         src_offset = 0;
                :         while (dst_offset < S) {
                :             // process group by group
                :             uint8_t* v_ptr = v + src_offset;
                :             auto v_f0 = reinterpret_cast<float*>(v_ptr);
                :             auto v_f1 = reinterpret_cast<float*>(v_ptr + src_stride);
                :             auto v_f2 = reinterpret_cast<float*>(v_ptr + 2 * src_stride);
                :             auto v_f3 = reinterpret_cast<float*>(v_ptr + 3 * src_stride);
                :             auto attn_w_vec0 = _mm512_set1_ps(weight[0] * v_f0[0]);
                :             auto attn_w_vec1 = _mm512_set1_ps(weight[1] * v_f1[0]);
                :             auto attn_w_vec2 = _mm512_set1_ps(weight[2] * v_f2[0]);
                :             auto attn_w_vec3 = _mm512_set1_ps(weight[3] * v_f3[0]);
                :             auto zp0 = _mm512_set1_ps(v_f0[1]);
                :             auto zp1 = _mm512_set1_ps(v_f1[1]);
                :             auto zp2 = _mm512_set1_ps(v_f2[1]);
                :             auto zp3 = _mm512_set1_ps(v_f3[1]);
                :             uint8_t* v_data_ptr = v + src_offset + params_offset;
                :             size_t i = 0;
                :             for (; i + vec_len_f32_avx512 <= group_size; i += vec_len_f32_avx512) {
                :                 auto v_out = mm512_uni_loadu_ps(out + dst_offset + i);
                :                 auto v0 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(
                :                                           _mm_loadu_si128(reinterpret_cast<__m128i*>(v_data_ptr + i)))),
                :                                         zp0);
                :                 auto v1 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(
                :                                           _mm_loadu_si128(reinterpret_cast<__m128i*>(v_data_ptr + i + src_stride)))),
                :                                         zp1);
                :                 auto v2 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(_mm_loadu_si128(
                :                                           reinterpret_cast<__m128i*>(v_data_ptr + i + 2 * src_stride)))),
                :                                         zp2);
                :                 auto v3 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(_mm_loadu_si128(
                :                                           reinterpret_cast<__m128i*>(v_data_ptr + i + 3 * src_stride)))),
                :                                         zp3);
                :                 v_out = _mm512_fmadd_ps(attn_w_vec0, v0, v_out);
                :                 v_out = _mm512_fmadd_ps(attn_w_vec1, v1, v_out);
                :                 v_out = _mm512_fmadd_ps(attn_w_vec2, v2, v_out);
                :                 v_out = _mm512_fmadd_ps(attn_w_vec3, v3, v_out);
                :                 _mm512_storeu_ps(out + dst_offset + i, v_out);
                :             }
                :             for (; i < group_size; i++) {
```

```
    302             :                 out[i] += weight[0] * (v_data_ptr[i] - v_f0[1]) * v_f0[0];
    303             :                 out[i] += weight[1] * (v_data_ptr[i + src_stride] - v_f1[1]) * v_f1[0];
    304             :                 out[i] += weight[2] * (v_data_ptr[i + 2 * src_stride] - v_f2[1]) * v_f2[0];
    305             :                 out[i] += weight[3] * (v_data_ptr[i + 3 * src_stride] - v_f3[1]) * v_f3[0];
    306             :             }
    307             :             dst_offset += group_size;
    308             :             src_offset += group_size + params_offset;
    309             :         }
    310             :         weight += 4;
    311             :         v += 4 * src_stride;
    312             :     }
    313             :     for (; j < block_size; j++) {
    314             :         dst_offset = 0;
    315             :         src_offset = 0;
    316             :         while (dst_offset < S) {
    317             :             uint8_t* v_ptr = v + src_offset;
    318             :             uint8_t* v_data_ptr = v_ptr + params_offset;
    319             :             auto v_f0 = reinterpret_cast<float*>(v_ptr);
    320             :             auto attn_w_vec0 = _mm512_set1_ps(weight[0] * v_f0[0]);
    321             :             auto zp0 = _mm512_set1_ps(v_f0[1]);
    322             :             size_t i = 0;
    323             :             for (; i + vec_len_f32_avx512 <= group_size; i += vec_len_f32_avx512) {
    324             :                 auto v_out = mm512_uni_loadu_ps((out + dst_offset + i));
    325             :                 auto v0 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(
    326             :                                         _mm_loadu_si128(reinterpret_cast<__m128i*>(v_data_ptr + i)))),
    327             :                                 zp0);
    328             :                 v_out = _mm512_fmadd_ps(attn_w_vec0, v0, v_out);
    329             :
    330             :                 _mm512_storeu_ps((out + dst_offset + i), v_out);
    331             :             }
    332             :             for (; i < group_size; i++) {
    333             :                 out[dst_offset + i] += weight[0] * (v_data_ptr[i] - v_f0[1]) * v_f0[0];
    334             :             }
    335             :             dst_offset += group_size;
    336             :             src_offset += group_size + params_offset;
    337             :         }
    338             :         v += src_stride;
    339             :         weight++;
    340             :     }
    341             :     return;
    342             : #    elif defined(HAVE_AVX2)
    343             :     size_t j = 0;
    344             :     for (; j < block_size; j++) {
    345             :         dst_offset = 0;
    346             :         src_offset = 0;
    347             :         while (dst_offset < S) {
    348             :             uint8_t* v_ptr = v + src_offset;
    349             :             uint8_t* v_data_ptr = v_ptr + params_offset;
    350             :             auto v_f0 = reinterpret_cast<float*>(v_ptr);
    351             :             auto attn_w_vec0 = _mm256_set1_ps(weight[0] * v_f0[0]);
    352             :             auto zp0 = _mm256_set1_ps(v_f0[1]);
    353             :             size_t i = 0;
    354             :             for (; i + vec_len_f32_avx2 <= group_size; i += vec_len_f32_avx2) {
    355             :                 auto v_out = mm256_uni_loadu_ps(out + dst_offset + i);
    356             :                 auto v0 = _mm256_sub_ps(_mm256_cvtepi32_ps(_mm256_cvtepu8_epi32(
    357             :                                         _mm_loadl_epi64(reinterpret_cast<__m128i*>(v_data_ptr + i)))),
    358             :                                 zp0);
    359             :                 v_out = _mm256_fmadd_ps(attn_w_vec0, v0, v_out);
    360             :
    361             :                 mm256_uni_storeu_ps(out + dst_offset + i, v_out);
    362             :             }
    363             :             for (; i < group_size; i++) {
    364             :                 out[dst_offset + i] += weight[0] * (v_data_ptr[i] - v_f0[1]) * v_f0[0];
    365             :             }
    366             :             dst_offset += group_size;
    367             :             src_offset += group_size + params_offset;
    368             :         }
    369             :         v += src_stride;
    370             :         weight++;
    371             :     }
    372             :     return;
    373             : #    elif defined(HAVE_SVE)
    374      136082 :     auto sve_pg = svptrue_b32();
    375      136082 :     size_t j = 0;
    376      521633 :     for (; j < block_size; ++j) {
    377             :         dst_offset = 0;
    378             :         src_offset = 0;
    379      816899 :         while (dst_offset < S) {
    380      431348 :             uint8_t* v_ptr = v + src_offset;
    381      431348 :             uint8_t* v_data_ptr = v_ptr + params_offset;
    382      431348 :             auto v_f0 = reinterpret_cast<float*>(v_ptr);
    383      431348 :             svfloat32_t attn_w_vec0 = svdup_n_f32(weight[0] * v_f0[0]);
    384      431348 :             svfloat32_t zp0 = svdup_n_f32(v_f0[1]);
    385      431348 :             size_t i = 0;
    386     6017628 :             for (; i + svcntw() <= group_size; i += svcntw()) {
    387     5586280 :                 auto v_out = svld1_f32(sve_pg, out + dst_offset + i);
    388     5586280 :                 svuint32_t reg1 = svld1ub_u32(sve_pg, v_data_ptr + i);
    389     5586280 :                 svfloat32_t reg2 = svcvt_f32_u32_z(sve_pg, reg1);
    390     5586280 :                 svfloat32_t v0 = svsub_f32_z(sve_pg, reg2, zp0);
    391     5586280 :                 v_out = svmla_f32_x(sve_pg, v_out, attn_w_vec0, v0);
    392     5586280 :                 svst1_f32(sve_pg, out + dst_offset + i, v_out);
    393             :             }
    394      431348 :             auto sve_pgt = svwhilelt_b32(i, group_size);
    395      431348 :             auto v_out = svld1_f32(sve_pgt, out + dst_offset + i);
    396      393161 :             svuint32_t reg1 = svld1ub_u32(sve_pgt, v_data_ptr + i);
    397      393161 :             svfloat32_t reg2 = svcvt_f32_u32_z(sve_pgt, reg1);
    398      393161 :             svfloat32_t v0 = svsub_f32_z(sve_pgt, reg2, zp0);
    399      393161 :             v_out = svmla_f32_x(sve_pgt, v_out, attn_w_vec0, v0);
    400      393161 :             svst1_f32(sve_pgt, out + dst_offset + i, v_out);
    401      424213 :             dst_offset += group_size;
    402      424213 :             src_offset += group_size + params_offset;
    403             :         }
    404      385551 :         v += src_stride;
    405      385551 :         weight++;
```

```
406             :     }
407             :     return;
408             : #    endif
409             :     for (size_t j = 0; j < block_size; j++) {
410             :         dst_offset = 0;
411             :         src_offset = 0;
412             :         while (dst_offset < S) {
413             :             auto v0 = reinterpret_cast<float*>(v + src_offset);
414             :             for (size_t i = 0; i < group_size; i++) {
415             :                 out[dst_offset + i] += weight[j] * (v[i + src_offset + params_offset] - v0[1]) * v0[0];
416             :             }
417             :             dst_offset += group_size;
418             :             src_offset += group_size + params_offset;
419             :         }
420             :         v += src_stride;
421             :     }
422             : }
423             :
424             : template <typename T, ov::element::Type_t SRC_PREC, std::enable_if_t<SRC_PREC == ov::element::u4, bool> = true>
425             : static void attn_acc_value_block(float* out,
426             :                                  float* weight,
427             :                                  void* v,
428             :                                  const size_t S,
429             :                                  const size_t block_size,
430             :                                  const size_t group_size) {
431             :     size_t src_offset = 0;
432             :     size_t dst_offset = 0;
433             :     const size_t params_offset = sizeof(float) * 2;
434             :     auto* v_ptr = reinterpret_cast<uint8_t*>(v);
435             :     auto sub_byte_multiplier = 8 / 4;
436             :     const size_t src_stride = S / group_size * (group_size / sub_byte_multiplier + params_offset);
437             :     auto extract_half_byte = [](uint8_t val, bool high_half) -> uint8_t {
438             :         uint8_t shift = high_half ? 0 : 4;
439             :
440             :         return static_cast<uint8_t>((val >> shift) & 0x000F);
441             :     };
442             :     for (size_t j = 0; j < block_size; j++) {
443             :         dst_offset = 0;
444             :         src_offset = 0;
445             :         while (dst_offset < S) {
446             :             auto v0 = reinterpret_cast<float*>(v_ptr + src_offset);
447             :             size_t i = 0;
448             : #    if defined(HAVE_AVX512F)
449             :             auto attn_w_vec0 = _mm512_set1_ps(weight[j] * v0[0]);
450             :             auto v_zp = _mm512_set1_ps(v0[1]);
451             :             for (; i + vec_len_f32_avx512 * 2 <= group_size; i += vec_len_f32_avx512 * 2) {
452             :                 auto data = _mm_loadu_si128(reinterpret_cast<__m128i*>(v_ptr + i / 2 + src_offset + params_offset));
453             :                 auto v_i32 = _mm512_cvtepu8_epi32(data);
454             :
455             :                 auto v_512_low_half = _mm512_srli_epi32(v_i32, 4);
456             :                 auto v_f32_low_half = _mm512_cvtepi32_ps(v_512_low_half);
457             :
458             :                 auto mask = _mm512_set1_epi32(0x0F);
459             :                 auto v_512_high_half = _mm512_and_si512(v_i32, mask);
460             :                 auto v_f32_high_half = _mm512_cvtepi32_ps(v_512_high_half);
461             :
462             :                 // q - zp
463             :                 v_f32_low_half = _mm512_sub_ps(v_f32_low_half, v_zp);
464             :                 v_f32_high_half = _mm512_sub_ps(v_f32_high_half, v_zp);
465             :
466             :                 __m512i idx1 = _mm512_set_epi32(23, 7, 22, 6, 21, 5, 20, 4, 19, 3, 18, 2, 17, 1, 16, 0);
467             :                 __m512i idx2 = _mm512_set_epi32(31, 15, 30, 14, 29, 13, 28, 12, 27, 11, 26, 10, 25, 9, 24, 8);
468             :                 __m512 first_half = _mm512_permutex2var_ps(v_f32_low_half, idx1, v_f32_high_half);
469             :                 __m512 second_half = _mm512_permutex2var_ps(v_f32_low_half, idx2, v_f32_high_half);
470             :                 auto v_out0 = mm512_uni_loadu_ps(out + dst_offset + i);
471             :                 auto v_out1 = mm512_uni_loadu_ps(out + dst_offset + i + vec_len_f32_avx512);
472             :                 v_out0 = _mm512_fmadd_ps(attn_w_vec0, first_half, v_out0);
473             :                 v_out1 = _mm512_fmadd_ps(attn_w_vec0, second_half, v_out1);
474             :                 mm512_uni_storeu_ps(out + dst_offset + i, v_out0);
475             :                 mm512_uni_storeu_ps(out + dst_offset + i + vec_len_f32_avx512, v_out1);
476             :             }
477             : #    elif defined(HAVE_AVX2)
478             :             auto v256_attn_w_vec0 = _mm256_set1_ps(weight[j] * v0[0]);
479             :             auto v256_zp = _mm256_set1_ps(v0[1]);
480             :             for (; i + vec_len_f32_avx2 * 2 <= group_size; i += vec_len_f32_avx2 * 2) {
481             :                 auto data = _mm_loadl_epi64(reinterpret_cast<__m128i*>(v_ptr + i / 2 + src_offset + params_offset));
482             :
483             :                 auto v_i32 = _mm256_cvtepu8_epi32(data);
484             :                 auto v_256_low_half = _mm256_srli_epi32(v_i32, 4);
485             :                 auto v_f32_low_half = _mm256_cvtepi32_ps(v_256_low_half);
486             :
487             :                 auto mask = _mm256_set1_epi32(0x0F);
488             :                 auto v_256_high_half = _mm256_and_si256(v_i32, mask);
489             :                 auto v_f32_high_half = _mm256_cvtepi32_ps(v_256_high_half);
490             :                 // q - zp
491             :                 v_f32_low_half = _mm256_sub_ps(v_f32_low_half, v256_zp);
492             :                 v_f32_high_half = _mm256_sub_ps(v_f32_high_half, v256_zp);
493             :
494             :                 auto v_out0 = mm256_uni_loadu_ps(out + dst_offset + i);
495             :                 auto v_out1 = mm256_uni_loadu_ps(out + dst_offset + i + vec_len_f32_avx2);
496             :
497             :                 __m256 first_half = _mm256_permute2f128_ps(v_f32_low_half, v_f32_high_half, 0x20);
498             :                 auto idx1 = _mm256_set_epi32(7, 3, 6, 2, 5, 1, 4, 0);
499             :                 first_half = _mm256_permutevar8x32_ps(first_half, idx1);
500             :                 __m256 second_half = _mm256_permute2f128_ps(v_f32_low_half, v_f32_high_half, 0x31);
501             :                 second_half = _mm256_permutevar8x32_ps(second_half, idx1);
502             :
503             :                 v_out0 = _mm256_fmadd_ps(v256_attn_w_vec0, first_half, v_out0);
504             :                 v_out1 = _mm256_fmadd_ps(v256_attn_w_vec0, second_half, v_out1);
505             :                 mm256_uni_storeu_ps(out + dst_offset + i, v_out0);
506             :                 mm256_uni_storeu_ps(out + dst_offset + i + vec_len_f32_avx2, v_out1);
507             :             }
508             : #    endif
509             :             for (; i < group_size; i += 2) {
```

```
                :                 uint8_t data = v_ptr[i / 2 + src_offset + params_offset];
                :                 float tmp0 = extract_half_byte(data, static_cast<bool>(i % 2));
                :                 float tmp1 = extract_half_byte(data, static_cast<bool>((i + 1) % 2));
                :                 out[dst_offset + i] += weight[j] * (tmp0 - v0[1]) * v0[0];
                :                 out[dst_offset + i + 1] += weight[j] * (tmp1 - v0[1]) * v0[0];
                :             }
                :             dst_offset += group_size;
                :             src_offset += group_size / sub_byte_multiplier + params_offset;
                :         }
                :         v_ptr += src_stride;
                :     }
                : }
                :
                : template <typename TA, typename TB>
                : static void dot_product_block(TA* a,
                :                               TB* b,
                :                               float* c,
                :                               const size_t n,
                :                               const size_t block_size,
                :                               const size_t group_size) {
                : #    if defined(HAVE_AVX512F)
                :     size_t j = 0;
                :     for (; j + 4 <= block_size; j += 4) {
                :         auto vsum0 = _mm512_setzero_ps();
                :         auto vsum1 = _mm512_setzero_ps();
                :         auto vsum2 = _mm512_setzero_ps();
                :         auto vsum3 = _mm512_setzero_ps();
                :         size_t i = 0;
                :         for (; i + vec_len_f32_avx512 <= n; i += vec_len_f32_avx512) {
                :             auto va = mm512_uni_loadu_ps(a + i);
                :             vsum0 = _mm512_fmadd_ps(va, mm512_uni_loadu_ps(b + i), vsum0);
                :             vsum1 = _mm512_fmadd_ps(va, mm512_uni_loadu_ps(b + i + n), vsum1);
                :             vsum2 = _mm512_fmadd_ps(va, mm512_uni_loadu_ps(b + i + 2 * n), vsum2);
                :             vsum3 = _mm512_fmadd_ps(va, mm512_uni_loadu_ps(b + i + 3 * n), vsum3);
                :         }
                :         float sum0 = _mm512_reduce_add_ps(vsum0);
                :         float sum1 = _mm512_reduce_add_ps(vsum1);
                :         float sum2 = _mm512_reduce_add_ps(vsum2);
                :         float sum3 = _mm512_reduce_add_ps(vsum3);
                :         for (; i < n; i++) {
                :             sum0 += a[i] * b[i];
                :             sum1 += a[i] * b[i + n];
                :             sum2 += a[i] * b[i + 2 * n];
                :             sum3 += a[i] * b[i + 3 * n];
                :         }
                :         c[0] = sum0;
                :         c[1] = sum1;
                :         c[2] = sum2;
                :         c[3] = sum3;
                :         c += 4;
                :         b += 4 * n;
                :     }
                :     for (; j < block_size; j++) {
                :         auto vsum = _mm512_setzero_ps();
                :         size_t i = 0;
                :         for (; i + vec_len_f32_avx512 <= n; i += vec_len_f32_avx512) {
                :             auto va = mm512_uni_loadu_ps(a + i);
                :             vsum = _mm512_fmadd_ps(va, mm512_uni_loadu_ps(b + i), vsum);
                :         }
                :         float sum = _mm512_reduce_add_ps(vsum);
                :         for (; i < n; i++) {
                :             sum += a[i] * b[i];
                :         }
                :         b += n;
                :         *c++ = sum;
                :     }
                :     return;
                : #    elif defined(HAVE_AVX2)
                :     size_t j = 0;
                :     for (; j + 4 <= block_size; j += 4) {
                :         auto vsum0 = _mm256_set1_ps(0.0f);
                :         auto vsum1 = _mm256_set1_ps(0.0f);
                :         auto vsum2 = _mm256_set1_ps(0.0f);
                :         auto vsum3 = _mm256_set1_ps(0.0f);
                :         size_t i = 0;
                :         for (; i + vec_len_f32_avx2 <= n; i += vec_len_f32_avx2) {
                :             auto va = mm256_uni_loadu_ps(a + i);
                :             vsum0 = _mm256_fmadd_ps(va, mm256_uni_loadu_ps(b + i), vsum0);
                :             vsum1 = _mm256_fmadd_ps(va, mm256_uni_loadu_ps(b + i + n), vsum1);
                :             vsum2 = _mm256_fmadd_ps(va, mm256_uni_loadu_ps(b + i + 2 * n), vsum2);
                :             vsum3 = _mm256_fmadd_ps(va, mm256_uni_loadu_ps(b + i + 3 * n), vsum3);
                :         }
                :         hsum(vsum0);
                :         hsum(vsum1);
                :         hsum(vsum2);
                :         hsum(vsum3);
                :         float sum0 = _mm256_cvtss_f32(vsum0);
                :         float sum1 = _mm256_cvtss_f32(vsum1);
                :         float sum2 = _mm256_cvtss_f32(vsum2);
                :         float sum3 = _mm256_cvtss_f32(vsum3);
                :         for (; i < n; i++) {
                :             sum0 += a[i] * b[i];
                :             sum1 += a[i] * b[i + n];
                :             sum2 += a[i] * b[i + 2 * n];
                :             sum3 += a[i] * b[i + 3 * n];
                :         }
                :         c[0] = sum0;
                :         c[1] = sum1;
                :         c[2] = sum2;
                :         c[3] = sum3;
                :         c += 4;
                :         b += 4 * n;
                :     }
                :     for (; j < block_size; j++) {
```

```
614              :         auto vsum = _mm256_set1_ps(0.0f);
615              :         size_t i = 0;
616              :         for (; i + vec_len_f32_avx2 <= n; i += vec_len_f32_avx2) {
617              :             auto va = mm256_uni_loadu_ps(a + i);
618              :             vsum = _mm256_fmadd_ps(va, mm256_uni_loadu_ps(b + i), vsum);
619              :         }
620              :         hsum(vsum);
621              :         float sum = _mm256_cvtss_f32(vsum);
622              :         for (; i < n; i++) {
623              :             sum += a[i] * b[i];
624              :         }
625              :         b += n;
626              :         *c++ = sum;
627              :     }
628              :     return;
629              : #    endif
630              :     for (size_t j = 0; j < block_size; j++) {
631              :         float sum = 0;
632              :         for (size_t i = 0; i < n; i++) {
633              :             sum += a[i] * b[i];
634              :         }
635              :         b += n;
636              :         *c++ = sum;
637              :     }
638              : }
639              :
640              : template <typename TA>
641            0 : static void dot_product_block_by_channel(TA* a, uint8_t* b, float* c, const size_t n, const size_t block_size) {
642            0 :     const size_t params_offset = sizeof(float) * 2 * n;
643            0 :     const size_t src_stride = n;
644            0 :     auto p_scales = reinterpret_cast<float*>(b);
645            0 :     auto p_zps = p_scales + n;
646            0 :     for (size_t j = 0; j < block_size; j++) {
647              :         float sum = 0.0f;
648              :         size_t i = 0;
649              : #    if defined(HAVE_AVX512F)
650              :         auto v512_sum0 = _mm512_set1_ps(0.0f);
651              :         auto v512_sum1 = _mm512_set1_ps(0.0f);
652              :         auto v512_sum2 = _mm512_set1_ps(0.0f);
653              :         auto v512_sum3 = _mm512_set1_ps(0.0f);
654              :         for (; i + 4 * vec_len_f32_avx512 <= n; i += vec_len_f32_avx512 * 4) {
655              :             auto v0_zp = _mm512_loadu_ps(p_zps + i);
656              :             auto v1_zp = _mm512_loadu_ps(p_zps + i + vec_len_f32_avx512);
657              :             auto v2_zp = _mm512_loadu_ps(p_zps + i + vec_len_f32_avx512 * 2);
658              :             auto v3_zp = _mm512_loadu_ps(p_zps + i + vec_len_f32_avx512 * 3);
659              :             auto v0_scale = _mm512_loadu_ps(p_scales + i);
660              :             auto v1_scale = _mm512_loadu_ps(p_scales + i + vec_len_f32_avx512);
661              :             auto v2_scale = _mm512_loadu_ps(p_scales + i + vec_len_f32_avx512 * 2);
662              :             auto v3_scale = _mm512_loadu_ps(p_scales + i + vec_len_f32_avx512 * 3);
663              :             auto va0 = mm512_uni_loadu_ps(a + i);
664              :             auto va1 = mm512_uni_loadu_ps(a + i + vec_len_f32_avx512);
665              :             auto va2 = mm512_uni_loadu_ps(a + i + vec_len_f32_avx512 * 2);
666              :             auto va3 = mm512_uni_loadu_ps(a + i + vec_len_f32_avx512 * 3);
667              :
668              :             auto vb0_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i));
669              :             auto vb1_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx512));
670              :             auto vb2_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx512 * 2));
671              :             auto vb3_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx512 * 3));
672              :
673              :             auto vb0_256 = _mm512_cvtepu8_epi32(vb0_128);
674              :             auto vb1_256 = _mm512_cvtepu8_epi32(vb1_128);
675              :             auto vb2_256 = _mm512_cvtepu8_epi32(vb2_128);
676              :             auto vb3_256 = _mm512_cvtepu8_epi32(vb3_128);
677              :
678              :             auto vb0 = _mm512_cvtepi32_ps(vb0_256);
679              :             auto vb1 = _mm512_cvtepi32_ps(vb1_256);
680              :             auto vb2 = _mm512_cvtepi32_ps(vb2_256);
681              :             auto vb3 = _mm512_cvtepi32_ps(vb3_256);
682              :
683              :             vb0 = _mm512_sub_ps(vb0, v0_zp);
684              :             vb1 = _mm512_sub_ps(vb1, v1_zp);
685              :             vb2 = _mm512_sub_ps(vb2, v2_zp);
686              :             vb3 = _mm512_sub_ps(vb3, v3_zp);
687              :
688              :             vb0 = _mm512_mul_ps(vb0, v0_scale);
689              :             vb1 = _mm512_mul_ps(vb1, v1_scale);
690              :             vb2 = _mm512_mul_ps(vb2, v2_scale);
691              :             vb3 = _mm512_mul_ps(vb3, v3_scale);
692              :
693              :             v512_sum0 = _mm512_fmadd_ps(va0, vb0, v512_sum0);
694              :             v512_sum1 = _mm512_fmadd_ps(va1, vb1, v512_sum1);
695              :             v512_sum2 = _mm512_fmadd_ps(va2, vb2, v512_sum2);
696              :             v512_sum3 = _mm512_fmadd_ps(va3, vb3, v512_sum3);
697              :         }
698              :         if (i + 2 * vec_len_f32_avx512 <= n) {
699              :             auto v0_zp = _mm512_loadu_ps(p_zps + i);
700              :             auto v1_zp = _mm512_loadu_ps(p_zps + i + vec_len_f32_avx512);
701              :             auto v0_scale = _mm512_loadu_ps(p_scales + i);
702              :             auto v1_scale = _mm512_loadu_ps(p_scales + i + vec_len_f32_avx512);
703              :
704              :             auto va0 = mm512_uni_loadu_ps(a + i);
705              :             auto va1 = mm512_uni_loadu_ps(a + i + vec_len_f32_avx512);
706              :
707              :             auto vb0_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i));
708              :             auto vb1_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx512));
709              :
710              :             auto vb0_256 = _mm512_cvtepu8_epi32(vb0_128);
711              :             auto vb1_256 = _mm512_cvtepu8_epi32(vb1_128);
712              :
713              :             auto vb0 = _mm512_cvtepi32_ps(vb0_256);
714              :             auto vb1 = _mm512_cvtepi32_ps(vb1_256);
715              :
716              :             vb0 = _mm512_sub_ps(vb0, v0_zp);
717              :             vb1 = _mm512_sub_ps(vb1, v1_zp);
```

```
718              :
719              :             vb0 = _mm512_mul_ps(vb0, v0_scale);
720              :             vb1 = _mm512_mul_ps(vb1, v1_scale);
721              :
722              :             v512_sum0 = _mm512_fmadd_ps(va0, vb0, v512_sum0);
723              :             v512_sum1 = _mm512_fmadd_ps(va1, vb1, v512_sum1);
724              :             i += 2 * vec_len_f32_avx512;
725              :         }
726              :         if (i + vec_len_f32_avx512 <= n) {
727              :             auto v0_zp = _mm512_loadu_ps(p_zps + i);
728              :             auto v0_scale = _mm512_loadu_ps(p_scales + i);
729              :
730              :             auto va0 = mm512_uni_loadu_ps(a + i);
731              :             auto vb0_128 = _mm_loadu_si128(reinterpret_cast<__m128i*>(b + params_offset + i));
732              :             auto vb0_256 = _mm512_cvtepu8_epi32(vb0_128);
733              :             auto vb0 = _mm512_cvtepi32_ps(vb0_256);
734              :             vb0 = _mm512_sub_ps(vb0, v0_zp);
735              :             vb0 = _mm512_mul_ps(vb0, v0_scale);
736              :             v512_sum0 = _mm512_fmadd_ps(va0, vb0, v512_sum0);
737              :             i += vec_len_f32_avx512;
738              :         }
739              :         v512_sum0 = _mm512_add_ps(v512_sum0, v512_sum1);
740              :         v512_sum2 = _mm512_add_ps(v512_sum2, v512_sum3);
741              :         v512_sum0 = _mm512_add_ps(v512_sum0, v512_sum2);
742              :         sum += _mm512_reduce_add_ps(v512_sum0);
743              : #    endif
744              : #    if defined(HAVE_AVX2)
745              :         auto vsum0 = _mm256_set1_ps(0.0f);
746              :         auto vsum1 = _mm256_set1_ps(0.0f);
747              :         auto vsum2 = _mm256_set1_ps(0.0f);
748              :         auto vsum3 = _mm256_set1_ps(0.0f);
749              :         for (; i + 4 * vec_len_f32_avx2 <= n; i += vec_len_f32_avx2 * 4) {
750              :             auto v0_zp = _mm256_loadu_ps(p_zps + i);
751              :             auto v1_zp = _mm256_loadu_ps(p_zps + i + vec_len_f32_avx2);
752              :             auto v2_zp = _mm256_loadu_ps(p_zps + i + vec_len_f32_avx2 * 2);
753              :             auto v3_zp = _mm256_loadu_ps(p_zps + i + vec_len_f32_avx2 * 3);
754              :             auto v0_scale = _mm256_loadu_ps(p_scales + i);
755              :             auto v1_scale = _mm256_loadu_ps(p_scales + i + vec_len_f32_avx2);
756              :             auto v2_scale = _mm256_loadu_ps(p_scales + i + vec_len_f32_avx2 * 2);
757              :             auto v3_scale = _mm256_loadu_ps(p_scales + i + vec_len_f32_avx2 * 3);
758              :
759              :             auto va0 = mm256_uni_loadu_ps(a + i);
760              :             auto va1 = mm256_uni_loadu_ps(a + i + vec_len_f32_avx2);
761              :             auto va2 = mm256_uni_loadu_ps(a + i + vec_len_f32_avx2 * 2);
762              :             auto va3 = mm256_uni_loadu_ps(a + i + vec_len_f32_avx2 * 3);
763              :
764              :             auto vb0_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i));
765              :             auto vb1_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx2));
766              :             auto vb2_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx2 * 2));
767              :             auto vb3_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx2 * 3));
768              :
769              :             auto vb0_256 = _mm256_cvtepu8_epi32(vb0_128);
770              :             auto vb1_256 = _mm256_cvtepu8_epi32(vb1_128);
771              :             auto vb2_256 = _mm256_cvtepu8_epi32(vb2_128);
772              :             auto vb3_256 = _mm256_cvtepu8_epi32(vb3_128);
773              :
774              :             auto vb0 = _mm256_cvtepi32_ps(vb0_256);
775              :             auto vb1 = _mm256_cvtepi32_ps(vb1_256);
776              :             auto vb2 = _mm256_cvtepi32_ps(vb2_256);
777              :             auto vb3 = _mm256_cvtepi32_ps(vb3_256);
778              :
779              :             vb0 = _mm256_sub_ps(vb0, v0_zp);
780              :             vb1 = _mm256_sub_ps(vb1, v1_zp);
781              :             vb2 = _mm256_sub_ps(vb2, v2_zp);
782              :             vb3 = _mm256_sub_ps(vb3, v3_zp);
783              :
784              :             vb0 = _mm256_mul_ps(vb0, v0_scale);
785              :             vb1 = _mm256_mul_ps(vb1, v1_scale);
786              :             vb2 = _mm256_mul_ps(vb2, v2_scale);
787              :             vb3 = _mm256_mul_ps(vb3, v3_scale);
788              :
789              :             vsum0 = _mm256_fmadd_ps(va0, vb0, vsum0);
790              :             vsum1 = _mm256_fmadd_ps(va1, vb1, vsum1);
791              :             vsum2 = _mm256_fmadd_ps(va2, vb2, vsum2);
792              :             vsum3 = _mm256_fmadd_ps(va3, vb3, vsum3);
793              :         }
794              :         if (i + 2 * vec_len_f32_avx2 <= n) {
795              :             auto v0_zp = _mm256_loadu_ps(p_zps + i);
796              :             auto v1_zp = _mm256_loadu_ps(p_zps + i + vec_len_f32_avx2);
797              :             auto v0_scale = _mm256_loadu_ps(p_scales + i);
798              :             auto v1_scale = _mm256_loadu_ps(p_scales + i + vec_len_f32_avx2);
799              :
800              :             auto va0 = mm256_uni_loadu_ps(a + i);
801              :             auto va1 = mm256_uni_loadu_ps(a + i + vec_len_f32_avx2);
802              :
803              :             auto vb0_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i));
804              :             auto vb1_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i + vec_len_f32_avx2));
805              :
806              :             auto vb0_256 = _mm256_cvtepu8_epi32(vb0_128);
807              :             auto vb1_256 = _mm256_cvtepu8_epi32(vb1_128);
808              :
809              :             auto vb0 = _mm256_cvtepi32_ps(vb0_256);
810              :             auto vb1 = _mm256_cvtepi32_ps(vb1_256);
811              :
812              :             vb0 = _mm256_sub_ps(vb0, v0_zp);
813              :             vb1 = _mm256_sub_ps(vb1, v1_zp);
814              :
815              :             vb0 = _mm256_mul_ps(vb0, v0_scale);
816              :             vb1 = _mm256_mul_ps(vb1, v1_scale);
817              :
818              :             vsum0 = _mm256_fmadd_ps(va0, vb0, vsum0);
819              :             vsum1 = _mm256_fmadd_ps(va1, vb1, vsum1);
820              :             i += 2 * vec_len_f32_avx2;
821              :         }
```

```
      822              :         if (i + vec_len_f32_avx2 <= n) {
      823              :             auto v0_zp = _mm256_loadu_ps(p_zps + i);
      824              :             auto v0_scale = _mm256_loadu_ps(p_scales + i);
      825              :
      826              :             auto va0 = mm256_uni_loadu_ps(a + i);
      827              :             auto vb0_128 = _mm_loadl_epi64(reinterpret_cast<__m128i*>(b + params_offset + i));
      828              :             auto vb0_256 = _mm256_cvtepu8_epi32(vb0_128);
      829              :             auto vb0 = _mm256_cvtepi32_ps(vb0_256);
      830              :
      831              :             vb0 = _mm256_sub_ps(vb0, v0_zp);
      832              :             vb0 = _mm256_mul_ps(vb0, v0_scale);
      833              :
      834              :             vsum0 = _mm256_fmadd_ps(va0, vb0, vsum0);
      835              :             i += vec_len_f32_avx2;
      836              :         }
      837              :         vsum0 = _mm256_add_ps(vsum0, vsum1);
      838              :         vsum2 = _mm256_add_ps(vsum2, vsum3);
      839              :         vsum0 = _mm256_add_ps(vsum0, vsum2);
      840              :         hsum(vsum0);
      841              :         sum += _mm256_cvtss_f32(vsum0);
      842              : #    endif
      843            0 :         for (; i < n; i++) {
      844            0 :             sum += a[i] * (b[params_offset + i] - p_zps[i]) * p_scales[i];
      845              :         }
      846            0 :         b += src_stride;
      847            0 :         *c++ = sum;
      848              :     }
      849            0 : }
      850              :
      851              : template <typename TA>
      852        90537 : static void dot_product_block(TA* a,
      853              :                               uint8_t* b,
      854              :                               float* c,
      855              :                               const size_t n,
      856              :                               const size_t block_size,
      857              :                               const size_t group_size) {
      858              :     // The layout for per token per head:
      859              :     // |scale(f32)|zeropoint(f32)|quantized feature(u8,idx_1)|quantized feature(u8,idx_2)|...|quantized
      860              :     // feature(u8,idx_S)| The quantized feature will start from 8bytes=sizeof(float)+sizeof(float)
      861        90537 :     size_t src_offset = 0;
      862        90537 :     size_t dst_offset = 0;
      863        90537 :     const size_t params_offset = sizeof(float) * 2;
      864        90537 :     const size_t src_stride = n / group_size * (group_size + params_offset);
      865              : #    if defined(HAVE_AVX512F)
      866              :     size_t j = 0;
      867              :     for (; j + 4 <= block_size; j += 4) {
      868              :         src_offset = 0;
      869              :         dst_offset = 0;
      870              :         float sum0 = 0.0f;
      871              :         float sum1 = 0.0f;
      872              :         float sum2 = 0.0f;
      873              :         float sum3 = 0.0f;
      874              :         while (dst_offset < n) {
      875              :             auto vsum0 = _mm512_setzero_ps();
      876              :             auto vsum1 = _mm512_setzero_ps();
      877              :             auto vsum2 = _mm512_setzero_ps();
      878              :             auto vsum3 = _mm512_setzero_ps();
      879              :             auto b0 = reinterpret_cast<float*>(b + src_offset);
      880              :             auto b1 = reinterpret_cast<float*>(b + src_offset + src_stride);
      881              :             auto b2 = reinterpret_cast<float*>(b + src_offset + src_stride * 2);
      882              :             auto b3 = reinterpret_cast<float*>(b + src_offset + src_stride * 3);
      883              :             auto v_zp0 = _mm512_set1_ps(b0[1]);
      884              :             auto v_zp1 = _mm512_set1_ps(b1[1]);
      885              :             auto v_zp2 = _mm512_set1_ps(b2[1]);
      886              :             auto v_zp3 = _mm512_set1_ps(b3[1]);
      887              :             size_t i = 0;
      888              :             uint8_t* b_data_ptr = b + src_offset + params_offset;
      889              :             for (; i + vec_len_f32_avx512 <= group_size; i += vec_len_f32_avx512) {
      890              :                 auto va = mm512_uni_loadu_ps(a + dst_offset + i);
      891              :                 auto vb0 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(
      892              :                                                 _mm_loadu_si128(reinterpret_cast<__m128i*>(b_data_ptr + i)))),
      893              :                                          v_zp0);
      894              :                 auto vb1 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(
      895              :                                                 _mm_loadu_si128(reinterpret_cast<__m128i*>(b_data_ptr + i + src_stride)))),
      896              :                                          v_zp1);
      897              :                 auto vb2 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(_mm_loadu_si128(
      898              :                                                 reinterpret_cast<__m128i*>(b_data_ptr + i + 2 * src_stride)))),
      899              :                                          v_zp2);
      900              :                 auto vb3 = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(_mm_loadu_si128(
      901              :                                                 reinterpret_cast<__m128i*>(b_data_ptr + i + 3 * src_stride)))),
      902              :                                          v_zp3);
      903              :
      904              :                 vsum0 = _mm512_fmadd_ps(va, vb0, vsum0);
      905              :                 vsum1 = _mm512_fmadd_ps(va, vb1, vsum1);
      906              :                 vsum2 = _mm512_fmadd_ps(va, vb2, vsum2);
      907              :                 vsum3 = _mm512_fmadd_ps(va, vb3, vsum3);
      908              :             }
      909              :             float group_sum0 = _mm512_reduce_add_ps(vsum0);
      910              :             float group_sum1 = _mm512_reduce_add_ps(vsum1);
      911              :             float group_sum2 = _mm512_reduce_add_ps(vsum2);
      912              :             float group_sum3 = _mm512_reduce_add_ps(vsum3);
      913              :             for (; i < group_size; i++) {
      914              :                 group_sum0 += a[i + dst_offset] * (b_data_ptr[i] - b0[1]);
      915              :                 group_sum1 += a[i + dst_offset] * (b_data_ptr[i + src_stride] - b1[1]);
      916              :                 group_sum2 += a[i + dst_offset] * (b_data_ptr[i + 2 * src_stride] - b2[1]);
      917              :                 group_sum3 += a[i + dst_offset] * (b_data_ptr[i + 3 * src_stride] - b3[1]);
      918              :             }
      919              :             sum0 += group_sum0 * b0[0];
      920              :             sum1 += group_sum1 * b1[0];
      921              :             sum2 += group_sum2 * b2[0];
      922              :             sum3 += group_sum3 * b3[0];
      923              :             dst_offset += group_size;
      924              :             src_offset += group_size + params_offset;
      925              :         }
```

```
926              :         c[0] = sum0;
927              :         c[1] = sum1;
928              :         c[2] = sum2;
929              :         c[3] = sum3;
930              :         c += 4;
931              :         b += 4 * src_stride;
932              :     }
933              :     for (; j < block_size; j++) {
934              :         src_offset = 0;
935              :         dst_offset = 0;
936              :         float sum = 0;
937              :         while (dst_offset < n) {
938              :             auto vsum = _mm512_setzero_ps();
939              :             auto b0 = reinterpret_cast<float*>(b + src_offset);
940              :             auto v_zp = _mm512_set1_ps(b0[1]);
941              :             size_t i = 0;
942              :             uint8_t* b_data_ptr = b + src_offset + params_offset;
943              :             for (; i + vec_len_f32_avx512 <= group_size; i += vec_len_f32_avx512) {
944              :                 auto va = mm512_uni_loadu_ps(a + dst_offset + i);
945              :                 auto vb = _mm512_sub_ps(_mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(
946              :                                                 _mm_loadu_si128(reinterpret_cast<__m128i*>(b_data_ptr + i)))),
947              :                                         v_zp);
948              :                 vsum = _mm512_fmadd_ps(va, vb, vsum);
949              :             }
950              :             float group_sum = _mm512_reduce_add_ps(vsum);
951              :             for (; i < group_size; i++) {
952              :                 group_sum += a[i + dst_offset] * (b_data_ptr[i] - b0[1]);
953              :             }
954              :             sum += group_sum * b0[0];
955              :             dst_offset += group_size;
956              :             src_offset += group_size + params_offset;
957              :         }
958              :         b += src_stride;
959              :         *c++ = sum;
960              :     }
961              :     return;
962              : #    elif defined(HAVE_AVX2)
963              :     size_t j = 0;
964              :     for (; j + 4 <= block_size; j += 4) {
965              :         src_offset = 0;
966              :         dst_offset = 0;
967              :         float sum0 = 0.0f;
968              :         float sum1 = 0.0f;
969              :         float sum2 = 0.0f;
970              :         float sum3 = 0.0f;
971              :         while (dst_offset < n) {
972              :             auto vsum0 = _mm256_setzero_ps();
973              :             auto vsum1 = _mm256_setzero_ps();
974              :             auto vsum2 = _mm256_setzero_ps();
975              :             auto vsum3 = _mm256_setzero_ps();
976              :             auto b0 = reinterpret_cast<float*>(b + src_offset);
977              :             auto b1 = reinterpret_cast<float*>(b + src_offset + src_stride);
978              :             auto b2 = reinterpret_cast<float*>(b + src_offset + src_stride * 2);
979              :             auto b3 = reinterpret_cast<float*>(b + src_offset + src_stride * 3);
980              :             auto v_zp0 = _mm256_set1_ps(b0[1]);
981              :             auto v_zp1 = _mm256_set1_ps(b1[1]);
982              :             auto v_zp2 = _mm256_set1_ps(b2[1]);
983              :             auto v_zp3 = _mm256_set1_ps(b3[1]);
984              :             size_t i = 0;
985              :             uint8_t* b_data_ptr = b + src_offset + params_offset;
986              :             for (; i + vec_len_f32_avx2 <= group_size; i += vec_len_f32_avx2) {
987              :                 auto va = mm256_uni_loadu_ps(a + dst_offset + i);
988              :                 auto vb0 = _mm256_sub_ps(_mm256_cvtepi32_ps(_mm256_cvtepu8_epi32(
989              :                                                 _mm_loadl_epi64(reinterpret_cast<__m128i*>(b_data_ptr + i)))),
990              :                                         v_zp0);
991              :                 auto vb1 = _mm256_sub_ps(_mm256_cvtepi32_ps(_mm256_cvtepu8_epi32(
992              :                                                 _mm_loadl_epi64(reinterpret_cast<__m128i*>(b_data_ptr + i + src_stride)))),
993              :                                         v_zp1);
994              :                 auto vb2 = _mm256_sub_ps(_mm256_cvtepi32_ps(_mm256_cvtepu8_epi32(_mm_loadl_epi64(
995              :                                                 reinterpret_cast<__m128i*>(b_data_ptr + i + 2 * src_stride)))),
996              :                                         v_zp2);
997              :                 auto vb3 = _mm256_sub_ps(_mm256_cvtepi32_ps(_mm256_cvtepu8_epi32(_mm_loadl_epi64(
998              :                                                 reinterpret_cast<__m128i*>(b_data_ptr + i + 3 * src_stride)))),
999              :                                         v_zp3);
1000             :
1001             :                 vsum0 = _mm256_fmadd_ps(va, vb0, vsum0);
1002             :                 vsum1 = _mm256_fmadd_ps(va, vb1, vsum1);
1003             :                 vsum2 = _mm256_fmadd_ps(va, vb2, vsum2);
1004             :                 vsum3 = _mm256_fmadd_ps(va, vb3, vsum3);
1005             :             }
1006             :             hsum(vsum0);
1007             :             hsum(vsum1);
1008             :             hsum(vsum2);
1009             :             hsum(vsum3);
1010             :             float group_sum0 = _mm256_cvtss_f32(vsum0);
1011             :             float group_sum1 = _mm256_cvtss_f32(vsum1);
1012             :             float group_sum2 = _mm256_cvtss_f32(vsum2);
1013             :             float group_sum3 = _mm256_cvtss_f32(vsum3);
1014             :             for (; i < group_size; i++) {
1015             :                 group_sum0 += a[dst_offset + i] * (b[i] - b0[1]);
1016             :                 group_sum1 += a[dst_offset + i] * (b[i + src_stride] - b1[1]);
1017             :                 group_sum2 += a[dst_offset + i] * (b[i + 2 * src_stride] - b2[1]);
1018             :                 group_sum3 += a[dst_offset + i] * (b[i + 3 * src_stride] - b3[1]);
1019             :             }
1020             :             sum0 += group_sum0 * b0[0];
1021             :             sum1 += group_sum1 * b1[0];
1022             :             sum2 += group_sum2 * b2[0];
1023             :             sum3 += group_sum3 * b3[0];
1024             :             dst_offset += group_size;
1025             :             src_offset += group_size + params_offset;
1026             :         }
1027             :         c[0] = sum0;
1028             :         c[1] = sum1;
1029             :         c[2] = sum2;
```

```
                :         c[3] = sum3;
                :         c += 4;
                :         b += 4 * src_stride;
                :     }
                :     for (; j < block_size; j++) {
                :         src_offset = 0;
                :         dst_offset = 0;
                :         float sum = 0;
                :         while (dst_offset < n) {
                :             auto vsum = _mm256_setzero_ps();
                :             auto b0 = reinterpret_cast<float*>(b + src_offset);
                :             auto v_zp = _mm256_set1_ps(b0[1]);
                :             size_t i = 0;
                :             uint8_t* b_data_ptr = b + src_offset + params_offset;
                :             for (; i + vec_len_f32_avx2 <= group_size; i += vec_len_f32_avx2) {
                :                 auto va = mm256_uni_loadu_ps(a + dst_offset + i);
                :                 auto vb = _mm256_sub_ps(_mm256_cvtepi32_ps(_mm256_cvtepu8_epi32(
                :                                             _mm_loadl_epi64(reinterpret_cast<__m128i*>(b_data_ptr + i)))),
                :                                         v_zp);
                :                 vsum = _mm256_fmadd_ps(va, vb, vsum);
                :             }
                :             hsum(vsum);
                :             float group_sum = _mm256_cvtss_f32(vsum);
                :             for (; i < group_size; i++) {
                :                 group_sum += a[i + dst_offset] * (b_data_ptr[i] - b0[1]);
                :             }
                :             sum += group_sum * b0[0];
                :             dst_offset += group_size;
                :             src_offset += group_size + params_offset;
                :         }
                :         b += src_stride;
                :         *c++ = sum;
                :     }
                :     return;
                : #    elif defined(HAVE_SVE)
          90537 :     auto scratch = svdup_f16_x(svptrue_b16(), 0);
          90537 :     size_t j = 0;
         575581 :     for (; j < block_size; j++) {
                :         src_offset = 0;
                :         dst_offset = 0;
                :         float sum = 0;
         988972 :         while (dst_offset < n) {
         503928 :             auto vsum = svdup_n_f32(0.0f);
         503928 :             auto b0 = reinterpret_cast<float*>(b + src_offset);
         503928 :             auto v_zp = svdup_n_f32(b0[1]);
         503928 :             size_t i = 0;
         503928 :             uint8_t* b_data_ptr = b + src_offset + params_offset;
         503928 :             auto vlen = svcntw();
                :             svbool_t pg_a, pg_f16;
        7847982 :             for (; i <= group_size; i += vlen) {
                :                 svfloat32_t va;
        7327884 :                 pg_a = svwhilelt_b32(i, group_size);
        7327884 :                 pg_f16 = svand_z(svptrue_b16(), svwhilelt_b16(svcnth() / 2, svcnth()), svwhilelt_b16(i, group_size));
                :                 if constexpr (std::is_same<TA, ov::float16>::value) {
        7327884 :                     auto load_src = svld1_f16(pg_f16, reinterpret_cast<float16_t*>(a + dst_offset + i));
        7344054 :                     auto src_interleave = svzip1_f16(load_src, scratch);
        7344054 :                     va = svcvt_f32_f16_z(pg_a, src_interleave);
                :                 } else {
              0 :                     va = svld1(pg_a, a + dst_offset + i);
                :                 }
        7344054 :                 auto r1 = svcvt_f32_u32_z(pg_a, svld1ub_u32(pg_a, b_data_ptr + i));
        7344054 :                 auto vb = svsub_f32_z(pg_a, r1, v_zp);
        7344054 :                 vsum = svmla_f32_m(pg_a, vsum, va, vb);
                :             }
         520098 :             float group_sum = svaddv_f32(svptrue_b32(), vsum);
         520098 :             sum += group_sum * b0[0];
         520098 :             dst_offset += group_size;
         520098 :             src_offset += group_size + params_offset;
                :         }
         485044 :         b += src_stride;
         485044 :         *c++ = sum;
                :     }
                :     return;
                : #    endif
                :     for (size_t j = 0; j < block_size; j++) {
                :         float sum = 0;
                :         dst_offset = 0;
                :         src_offset = 0;
                :         while (dst_offset < n) {
                :             auto b0 = reinterpret_cast<float*>(b + src_offset);
                :             float group_sum = 0.0f;
                :             for (size_t i = 0; i < group_size; i++) {
                :                 group_sum += a[dst_offset + i] * (b[src_offset + params_offset + i] - b0[1]);
                :             }
                :             sum += group_sum * b0[0];
                :             dst_offset += group_size;
                :             src_offset += group_size + params_offset;
                :         }
                :         b += src_stride;
                :         *c++ = sum;
                :     }
                : }
                :
                : template <typename T>
          59507 : static void attn_reduce(T* dst, float* temp, size_t M, size_t S, size_t temp_stride) {
          59507 :     size_t i = 0;
                : #    if defined(HAVE_AVX512F)
                :     for (; i + vec_len_f32_avx512 <= S; i += vec_len_f32_avx512) {
                :         auto* src = temp + i;
                :         auto result_vec_fp32 = _mm512_setzero_ps();
                :         for (size_t m = 0; m < M; m++) {
                :             auto o_vec_fp32 = _mm512_loadu_ps(src);
                :             result_vec_fp32 = _mm512_add_ps(result_vec_fp32, o_vec_fp32);
                :             src += temp_stride;
```

```
1134             :         }
1135             :         // save to bf16
1136             :         mm512_uni_storeu_ps(dst + i, result_vec_fp32);
1137             :     }
1138             : #    elif defined(HAVE_AVX2)
1139             :     for (; i + vec_len_f32_avx2 <= S; i += vec_len_f32_avx2) {
1140             :         auto* src = temp + i;
1141             :         auto result_vec_fp32 = _mm256_set1_ps(0.0f);
1142             :         for (size_t m = 0; m < M; m++) {
1143             :             auto o_vec_fp32 = mm256_uni_loadu_ps(src);
1144             :             result_vec_fp32 = _mm256_add_ps(result_vec_fp32, o_vec_fp32);
1145             :             src += temp_stride;
1146             :         }
1147             :         mm256_uni_storeu_ps(dst + i, result_vec_fp32);
1148             :     }
1149             : #    endif
1150      616341 :     for (; i < S; i++) {
1151      521448 :         auto* src = temp + i;
1152      521448 :         float sum = 0.0f;
1153             :         // sum result from all threads partition
1154    31818536 :         for (size_t m = 0; m < M; m++) {
1155    31297088 :             sum += src[0];
1156    31297088 :             src += temp_stride;
1157             :         }
1158      521448 :         dst[i] = sum;
1159             :     }
1160       94893 : }
1161             :
1162             : // N must be multiple of 16
1163             : template <typename TDST,
1164             :           ov::element::Type_t SRC_PREC,
1165             :           std::enable_if_t<(SRC_PREC != ov::element::u8 && SRC_PREC != ov::element::u4), bool> = true>
1166        3615 : void transpose_16NxK(TDST* dst,
1167             :                      void* src,
1168             :                      TDST* tmp,
1169             :                      const size_t N,
1170             :                      const size_t K,
1171             :                      const size_t dst_stride,
1172             :                      const size_t src_stride,
1173             :                      const size_t group_size,
1174             :                      const bool quant_key_bychannel) {
1175        3615 :     size_t k = 0;
1176        3615 :     auto* src_ptr = reinterpret_cast<typename ov::element_type_traits<SRC_PREC>::value_type*>(src);
1177       33109 :     for (; k + 16 <= K; k += 16) {
1178       74231 :         for (size_t n = 0; n < N; n += 16) {
1179       44737 :             transpose_16x16_kernel(dst + n, src_ptr + n * src_stride, dst_stride, src_stride);
1180             :         }
1181             :
1182       29494 :         dst += 16 * dst_stride;
1183       29494 :         src_ptr += 16;
1184             :     }
1185        4029 :     if (k < K) {
1186           0 :         for (size_t n = 0; n < N; n += 16) {
1187           0 :             transpose_16xK_kernel(dst + n, src_ptr + n * src_stride, K - k, dst_stride, src_stride);
1188             :         }
1189             :     }
1190        4029 : }
1191             : #    if defined(HAVE_AVX512F)
1192             : template <typename T,
1193             :           ov::element::Type_t SRC_PREC,
1194             :           typename std::enable_if<(SRC_PREC == ov::element::bf16 || SRC_PREC == ov::element::f16) &&
1195             :                                   (SRC_PREC == precision_of<T>::value),
1196             :                                bool>::type = true>
1197             : static void transpose_16NxK(T* dst,
1198             :                             T* src,
1199             :                             T* tmp,
1200             :                             const size_t N,
1201             :                             const size_t K,
1202             :                             const size_t dst_stride,
1203             :                             const size_t src_stride,
1204             :                             const size_t group_size,
1205             :                             const bool quant_key_bychannel) {
1206             :     // will treat as uint32_t transpose
1207             :     auto s = reinterpret_cast<uint32_t*>(src);
1208             :     auto d = reinterpret_cast<uint32_t*>(dst);
1209             :     transpose_16NxK<uint32_t, ov::element::u32>(d,
1210             :                                                 s,
1211             :                                                 reinterpret_cast<uint32_t*>(0),
1212             :                                                 N,
1213             :                                                 K >> 1,
1214             :                                                 dst_stride,
1215             :                                                 src_stride >> 1,
1216             :                                                 group_size,
1217             :                                                 false);
1218             : }
1219             : #    endif
1220             :
1221             : template <typename TDST, ov::element::Type_t SRC_PREC, std::enable_if_t<SRC_PREC == ov::element::u8, bool> = true>
1222        3521 : void transpose_16NxK(TDST* dst,
1223             :                      void* src,
1224             :                      TDST* tmp,
1225             :                      const size_t N,
1226             :                      const size_t K,
1227             :                      const size_t dst_stride,
1228             :                      const size_t src_stride,
1229             :                      const size_t group_size,
1230             :                      const bool quant_key_bychannel) {
1231             :     // The layout for per token per head:
1232             :     // |scale(f32)|zeropoint(f32)|quantized feature(u8,idx_1)|quantized feature(u8,idx_2)|...|quantized
1233             :     // feature(u8,idx_S)| The quantized feature will start from 8bytes=sizeof(float)+sizeof(float)
1234        3521 :     auto s = reinterpret_cast<typename ov::element_type_traits<SRC_PREC>::value_type*>(src);
1235        3521 :     auto t = tmp;
1236             :     // if group_size not set, the whole row is used as a group
1237        3521 :     if (quant_key_bychannel) {
```

```
1238          0 :         auto p_scales = reinterpret_cast<float*>(s);
1239          0 :         auto p_zps = p_scales + K;
1240          0 :         s = s + sizeof(float) * 2 * K;
1241          0 :         attn_dequant_u8_by_channel_kernel(s, t, N, K, K, src_stride, p_scales, p_zps);
1242            :     } else {
1243      37331 :         for (size_t n = 0; n < N; n++) {
1244            :             size_t src_offset = 0;
1245            :             size_t dst_offset = 0;
1246      67723 :             while (dst_offset < K) {
1247      33913 :                 auto f = reinterpret_cast<float*>(s + src_offset);
1248      33913 :                 attn_dequant_kernel<TDST, SRC_PREC>(s + src_offset + sizeof(float) * 2,
1249      33913 :                                                     t + dst_offset,
1250            :                                                     group_size,
1251            :                                                     f[0],
1252            :                                                     f[1]);
1253      33802 :                 src_offset += group_size + sizeof(float) * 2;
1254      33802 :                 dst_offset += group_size;
1255            :             }
1256      33810 :             s += src_offset;
1257      33810 :             t += src_stride;
1258            :         }
1259            :     }
1260       3410 :     transpose_16NxK<TDST, precision_of<TDST>::value>(dst,
1261            :                                                      tmp,
1262            :                                                      reinterpret_cast<TDST*>(0),
1263            :                                                      N,
1264            :                                                      K,
1265            :                                                      dst_stride,
1266            :                                                      src_stride,
1267            :                                                      0,
1268            :                                                      false);
1269       4032 : }
1270            :
1271            : // dequant f16/u8 to float
1272            : template <typename T,
1273            :           ov::element::Type_t SRC_PREC,
1274            :           std::enable_if_t<SRC_PREC != ov::element::u8 && precision_of<T>::value == SRC_PREC, bool> = true>
1275            : static inline void dequant(T* dst, void* src, const size_t N, const size_t K, const size_t group_size) {
1276            :     // never called
1277            :     OPENVINO_THROW("dequant: should not be called.");
1278            : }
1279            : template <typename T, ov::element::Type_t SRC_PREC, std::enable_if_t<SRC_PREC == ov::element::f16, bool> = true>
1280            : static inline void dequant(float* dst, void* src, const size_t N, const size_t K, const size_t group_size) {
1281            :     cvt_copy(dst, reinterpret_cast<ov::float16*>(src), 1, K * N, 0, 0);
1282            : }
1283            :
1284            : template <typename TDST,
1285            :           ov::element::Type_t SRC_PREC,
1286            :           std::enable_if_t<SRC_PREC == ov::element::u4 || SRC_PREC == ov::element::u8, bool> = true>
1287       4032 : void dequant(TDST* dst, uint8_t* src, const size_t N, const size_t K, const size_t group_size) {
1288            :     // The layout for per token per head:
1289            :     // |scale(f32)|zeropoint(f32)|quantized feature(u8,idx_1)|quantized feature(u8,idx_2)|...|quantized
1290            :     // feature(u8,idx_S)| The quantized feature will start from 8bytes=sizeof(float)+sizeof(float)
1291       4032 :     auto s = src;
1292       4032 :     const size_t params_offset = sizeof(float) * 2;
1293       4032 :     const size_t sub_byte_mulitplier = get_sub_byte_multiplier(SRC_PREC);
1294            :
1295      40996 :     for (size_t n = 0; n < N; n++) {
1296            :         size_t src_offset = 0;
1297            :         size_t dst_offset = 0;
1298      74353 :         while (dst_offset < K) {
1299      37388 :             auto f = reinterpret_cast<float*>(s + src_offset);
1300      37388 :             attn_dequant_kernel<TDST, SRC_PREC>(s + src_offset + params_offset,
1301      37388 :                                                 dst + dst_offset,
1302            :                                                 group_size,
1303            :                                                 f[0],
1304            :                                                 f[1]);
1305      36949 :             src_offset += group_size / sub_byte_mulitplier + params_offset;
1306      36949 :             dst_offset += group_size;
1307            :         }
1308      36965 :         s += src_offset;
1309      36965 :         dst += K;
1310            :     }
1311       3592 : }
1312            :
1313            : #    if defined(HAVE_AVX512F)
1314            : template <typename T,
1315            :           typename = typename std::
1316            :               enable_if<(std::is_same<T, ov::bfloat16>::value || std::is_same<T, ov::float16>::value), bool>::type>
1317            : static void pack_32x32_kernel(T* dst, T* src, size_t dst_stride, size_t src_stride) {
1318            :     static const uint64_t idx[8] = {0, 4, 1, 5, 2, 6, 3, 7};
1319            :     auto midx = _mm512_loadu_si512(idx);
1320            :     for (size_t i = 0; i < 16; i++) {
1321            :         auto a = _mm512_loadu_si512(src);  // [a1  a2  a3 a4 | a5  a6  a7 a8]   total 512-bits in 8 64bits unit
1322            :         auto b = _mm512_loadu_si512(src + src_stride);  // [b1  b2  b3 b4 | b5  b6  b7 b8]   total 512-bits
1323            :         a = _mm512_permutexvar_epi64(midx, a);          // [a1 a5 | a2 a6 | a3 a7 | a4 a8]
1324            :         b = _mm512_permutexvar_epi64(midx, b);          // [b1 b5 | b2 b6 | b3 b7 | b4 b8]
1325            :         auto B0 = _mm512_unpacklo_epi16(
1326            :             a,
1327            :             b);  // [ a1&b1  a2&b2   a3&b3   a4&b4] for each 128-bits lane, interleave word in low 64 bits
1328            :         auto B1 = _mm512_unpackhi_epi16(
1329            :             a,
1330            :             b);  // [ a5&b5  a6&b6   a7&b7   a8&b8] for each 128-bits lane, interleave word in high 64 bits
1331            :         _mm512_storeu_si512(dst, B0);
1332            :         _mm512_storeu_si512(dst + 32, B1);
1333            :         src += 2 * src_stride;
1334            :         dst += 2 * dst_stride;
1335            :     }
1336            : }
1337            :
1338            : template <typename T,
1339            :           typename = typename std::
1340            :               enable_if<(std::is_same<T, ov::bfloat16>::value || std::is_same<T, ov::float16>::value), bool>::type>
1341            : static void pack_32x16_kernel(T* dst, T* src, size_t dst_stride, size_t src_stride) {
```

```
1342              :     static const uint64_t idx[8] = {0, 4, 1, 5, 2, 6, 3, 7};
1343              :     auto midx = _mm512_loadu_si512(idx);
1344              :     for (size_t i = 0; i < 16; i++) {
1345              :         auto x =
1346              :             _mm256_loadu_si256(reinterpret_cast<__m256i*>(src));  // [a1  a2  a3 a4]   total 256-bits in 4 64bits unit
1347              :         auto y = _mm256_loadu_si256(reinterpret_cast<__m256i*>(src + src_stride));  // [b1  b2  b3 b4]   total 256-bits
1348              :         auto a = _mm512_castsi256_si512(x);
1349              :         auto b = _mm512_castsi256_si512(y);
1350              :         a = _mm512_permutexvar_epi64(midx, a);  // [a1 x | a2 x | a3 x | a4 x]
1351              :         b = _mm512_permutexvar_epi64(midx, b);  // [b1 x | b2 x | b3 x | b4 x]
1352              :         auto B0 = _mm512_unpacklo_epi16(a, b);
1353              :         _mm512_storeu_si512(dst, B0);
1354              :         src += 2 * src_stride;
1355              :         dst += 2 * dst_stride;
1356              :     }
1357              : }
1358              :
1359              : template <typename T,
1360              :           typename = typename std::
1361              :               enable_if<(std::is_same<T, ov::bfloat16>::value || std::is_same<T, ov::float16>::value), bool>::type>
1362              : static void pack_32xK_kernel(T* dst, T* src, size_t dst_stride, size_t src_stride, size_t K) {
1363              :     static const uint64_t idx[8] = {0, 4, 1, 5, 2, 6, 3, 7};
1364              :     auto midx = _mm512_loadu_si512(idx);
1365              :     __mmask16 mask = (1 << K) - 1;
1366              :     for (size_t i = 0; i < 16; i++) {
1367              :         auto x = _mm256_maskz_loadu_epi16(mask, src);               // [a1  a2  a3 a4]   total 256-bits in 4 64bits unit
1368              :         auto y = _mm256_maskz_loadu_epi16(mask, src + src_stride);  // [b1  b2  b3 b4]   total 256-bits
1369              :         auto a = _mm512_castsi256_si512(x);
1370              :         auto b = _mm512_castsi256_si512(y);
1371              :         a = _mm512_permutexvar_epi64(midx, a);  // [a1 x | a2 x | a3 x | a4 x]
1372              :         b = _mm512_permutexvar_epi64(midx, b);  // [b1 x | b2 x | b3 x | b4 x]
1373              :         auto B0 = _mm512_unpacklo_epi16(a, b);
1374              :         _mm512_mask_storeu_epi32(dst, mask, B0);
1375              :         src += 2 * src_stride;
1376              :         dst += 2 * dst_stride;
1377              :     }
1378              : }
1379              :
1380              : template <typename TDST,
1381              :           ov::element::Type_t SRC_PREC,
1382              :           typename std::enable_if<precision_of<TDST>::value != ov::element::f32 &&
1383              :                                   (SRC_PREC == ov::element::bf16 || SRC_PREC == ov::element::f16),
1384              :                                   bool>::type = true>
1385              : static void pack_32NxK(TDST* dst,
1386              :                        void* src,
1387              :                        TDST* tmp,
1388              :                        const size_t N,
1389              :                        const size_t K,
1390              :                        const size_t dst_stride,
1391              :                        const size_t src_stride,
1392              :                        const size_t group_size) {
1393              :     auto src_ptr = reinterpret_cast<typename ov::element_type_traits<SRC_PREC>::value_type*>(src);
1394              :     for (size_t n = 0; n < N; n += 32) {
1395              :         size_t k = 0;
1396              :         for (; k + 32 <= K; k += 32) {
1397              :             pack_32x32_kernel(dst + k * 2, src_ptr + k, dst_stride, src_stride);
1398              :         }
1399              :         if (k + 16 <= K) {
1400              :             pack_32x16_kernel(dst + k * 2, src_ptr + k, dst_stride, src_stride);
1401              :             k += 16;
1402              :         }
1403              :         if (k < K) {
1404              :             pack_32xK_kernel(dst + k * 2, src_ptr + k, dst_stride, src_stride, K - k);
1405              :         }
1406              :
1407              :         dst += 32 * dst_stride;
1408              :         src_ptr += 32 * src_stride;
1409              :     }
1410              : }
1411              :
1412              : template <typename TDST,
1413              :           ov::element::Type_t SRC_PREC,
1414              :           typename std::enable_if<precision_of<TDST>::value != ov::element::f32 &&
1415              :                                   (SRC_PREC == ov::element::u4 || SRC_PREC == ov::element::u8),
1416              :                                   bool>::type = true>
1417              : static void pack_32NxK(TDST* dst,
1418              :                        void* src,
1419              :                        TDST* tmp,
1420              :                        const size_t N,
1421              :                        const size_t K,
1422              :                        const size_t dst_stride,
1423              :                        const size_t src_stride,
1424              :                        const size_t group_size) {
1425              :     // The layout for per token per head:
1426              :     // |scale(f32)|zeropoint(f32)|quantized feature(u8,idx_1)|quantized feature(u8,idx_2)|...|quantized
1427              :     // feature(u8,idx_S)| The quantized feature will start from 8bytes=sizeof(float)+sizeof(float)
1428              :     auto s = reinterpret_cast<uint8_t*>(src);
1429              :     auto t = tmp;
1430              :     // if group_size not set, the whole row is used as a group
1431              :     const size_t sub_byte_mulitplier = get_sub_byte_multiplier(SRC_PREC);
1432              :     for (size_t n = 0; n < N; n++) {
1433              :         size_t src_offset = 0;
1434              :         size_t dst_offset = 0;
1435              :         while (dst_offset < K) {
1436              :             auto f = reinterpret_cast<float*>(s + src_offset);
1437              :             attn_dequant_kernel<TDST, SRC_PREC>(s + (src_offset + sizeof(float) * 2),
1438              :                                                 t + dst_offset,
1439              :                                                 group_size,
1440              :                                                 f[0],
1441              :                                                 f[1]);
1442              :             src_offset += group_size / sub_byte_mulitplier + sizeof(float) * 2;
1443              :             dst_offset += group_size;
1444              :         }
1445              :         s += src_offset;
```

```
1446             :         t += src_stride;
1447             :     }
1448             :     pack_32NxK<TDST, precision_of<TDST>::value>(dst,
1449             :                                                 tmp,
1450             :                                                 reinterpret_cast<TDST*>(0),
1451             :                                                 N,
1452             :                                                 K,
1453             :                                                 dst_stride,
1454             :                                                 src_stride,
1455             :                                                 group_size);
1456             : }
1457             : #    endif
1458             :
1459             : template <typename TDST,
1460             :           ov::element::Type_t SRC_PREC,
1461             :           std::enable_if_t<precision_of<TDST>::value == ov::element::f32, bool> = true>
1462             : static void pack_32NxK(TDST* dst,
1463             :                        void* src,
1464             :                        TDST* tmp,
1465             :                        const size_t N,
1466             :                        const size_t K,
1467             :                        const size_t dst_stride,
1468             :                        const size_t src_stride,
1469             :                        const size_t group_size) {
1470             :     // never called
1471             :     OPENVINO_THROW("pack_32NxK: should not be called.");
1472             : }
1473             :
1474             : template <class T>
1475           0 : void fill_rotation_coefficients_from_lut(T* rotation_coefficients_block_data,
1476             :                                          const int32_t* rotation_deltas_block_data,
1477             :                                          size_t rotation_deltas_token_stride,
1478             :                                          const T* rotation_trig_lut,
1479             :                                          size_t block_size,
1480             :                                          size_t embedding_size) {
1481           0 :     size_t dst_offset = 0;
1482           0 :     for (size_t tok_idx = 0; tok_idx < block_size; tok_idx++) {
1483           0 :         size_t gather_idx = *(rotation_deltas_block_data + rotation_deltas_token_stride * tok_idx);
1484           0 :         size_t src_offset = gather_idx * embedding_size;
1485           0 :         std::memcpy(rotation_coefficients_block_data + dst_offset,
1486           0 :                     rotation_trig_lut + src_offset,
1487             :                     embedding_size * sizeof(T));
1488           0 :         dst_offset += embedding_size;
1489             :     }
1490           0 : }
1491             :
1492             : template <class KVCACHE_TYPE>
1493           0 : void rotate_kv_cache(PlainTensor& key_cache,
1494             :                      const PlainTensor& rotated_block_indices,
1495             :                      const PlainTensor& rotation_deltas,
1496             :                      const PlainTensor& rotation_trig_lut,
1497             :                      PlainTensor& rotation_coefficients_scratch) {
1498           0 :     size_t num_blocks_in_total = key_cache.size(0);
1499           0 :     size_t num_heads = key_cache.size(1);  // H;
1500           0 :     size_t block_size = key_cache.size(2);
1501           0 :     size_t embedding_size = key_cache.size(3);  // S;
1502             :
1503           0 :     size_t num_rotated_blocks = rotated_block_indices.size(0);
1504           0 :     auto* rotated_block_indices_data = rotated_block_indices.ptr<int32_t>();
1505           0 :     auto* rotation_trig_lut_data = rotation_trig_lut.ptr<float>();
1506             :
1507           0 :     size_t rotation_deltas_token_stride = 0;
1508           0 :     size_t rotation_deltas_block_stride = 1;
1509             :
1510           0 :     bool is_per_token = (rotation_deltas.shape()[1] == block_size);
1511           0 :     if (is_per_token) {
1512           0 :         rotation_deltas_token_stride = 1;
1513           0 :         rotation_deltas_block_stride = block_size;
1514             :     }
1515             :
1516           0 :     for (size_t i = 0; i < num_rotated_blocks; i++) {
1517           0 :         size_t rotated_block_index = *(rotated_block_indices_data + i);
1518           0 :         OPENVINO_ASSERT(rotated_block_index < num_blocks_in_total);
1519             :
1520           0 :         int32_t* rotation_deltas_block_data = rotation_deltas.ptr<int32_t>() + i * rotation_deltas_block_stride;
1521             :
1522           0 :         auto* rotation_coefficient_block_data = rotation_coefficients_scratch.ptr<float>();
1523           0 :         fill_rotation_coefficients_from_lut(rotation_coefficient_block_data,
1524             :                                             rotation_deltas_block_data,
1525             :                                             rotation_deltas_token_stride,
1526             :                                             rotation_trig_lut_data,
1527             :                                             block_size,
1528             :                                             embedding_size);
1529           0 :         auto* cache_block_ptr = key_cache.ptr<KVCACHE_TYPE>(rotated_block_index);
1530           0 :         rotate_kv_cache_block(cache_block_ptr, rotation_coefficient_block_data, num_heads, block_size, embedding_size);
1531             :     }
1532           0 : }
1533             :
1534             : template <typename DATA_TYPE, typename KEY_CACHE_TYPE, ov::element::Type_t VALUE_PREC>
1535             : struct MHAHelper {
1536             :     // initialize once
1537             :     size_t H;
1538             :     size_t S;
1539             :     size_t SV;
1540             :     size_t Hk;
1541             :     size_t _h_each_group_len;
1542             :     size_t _block_size;
1543             :     size_t _nthr;
1544             :     size_t _sliding_window;
1545             :     float _d_scale;
1546             :     size_t _key_group_size = 0;
1547             :     size_t _value_group_size = 0;
1548             :     bool _quant_key_bychannel = 0;
1549             :     bool AarchF16 = false;
```

```
                :
                :     PlainTensor _weight;        // [nthr, H, 32, rnd_up(kv_len, block_size)], shared by first and second loop along bh
                :     PlainTensor _output;        // [nthr, 32, H, S], shared by first and second loop along bh
                :     PlainTensor _qk_scratch_a;  // [nthr, scratch_a_size]
                :     PlainTensor _qk_scratch_b;  // [B, rnd_up(kv_len, block_size), Hk, scratch_b_size]
                :     PlainTensor _wv_scratch_a;
                :     PlainTensor _wv_scratch_b;
                :     PlainTensor _alibi_lookup;
                :     PlainTensor _score_output;
                :     std::vector<size_t> _wsp;
                :     size_t _wsp_size_per_thread = 0;
                :
                :     std::vector<std::shared_ptr<BrgemmKernel>> _qk_gemm;
                :     std::vector<std::shared_ptr<BrgemmKernel>> _wv_gemm;
                :     // will accumulate C buffer
                :     std::vector<std::shared_ptr<BrgemmKernel>> _wv_gemm_acc;
                : // second token
                : #    if defined(OPENVINO_ARCH_X86_64)
                :     std::shared_ptr<JitMatMulVecAMX> _gemv;
                : #    endif
                :     ov::element::Type _fastpath_valid_prec = ov::element::dynamic;
                :     // second token for bhl loop
                :     PlainTensor _weight_bhl;
                :     PlainTensor _output_bhl;
                :     PlainTensor _score_offsets_aligned;
                :     PlainTensor _score_offsets;
                :
                :     PlainTensor _block_rotation_coefficient_scratch;
                :
                :     MHAHelper() {
                :         _weight.resize<float>({size_t{1}, size_t{1}, size_t{1}, size_t{1}});
                :     }
                :
              32 :     explicit MHAHelper(size_t key_group_size, size_t value_group_size, bool quant_key_bychannel)
              32 :         : _key_group_size(key_group_size),
              32 :           _value_group_size(value_group_size),
              32 :           _quant_key_bychannel(quant_key_bychannel) {
              64 :         _weight.resize<float>({size_t{1}, size_t{1}, size_t{1}, size_t{1}});
              32 :     }
                :
            1280 :     void init(size_t H,
                :               size_t S,
                :               size_t SV,
                :               size_t Hk,
                :               size_t h_each_group_len,
                :               size_t block_size,
                :               size_t sliding_window,
                :               float d_scale,
                :               size_t kv_len,
                :               bool init_alibi_lookup,
                :               bool init_rotation_coefficient_scratch) {
                :         // query shape: [B, H, L, S]
                :         // present_key shape: [block, H, 32, S]
                :         // Q*K': [M1, S] * [M2, S]'
                :         //   kernel: Q:[1~block_size, S] * K':[block_size, S]'
                :         //   aka: M:1~block_size, N:block_size, K:S
                :         // (Q*K')*V: [M1, M2] * [M2, S]
                :         //   kernel: (Q*K'):[1~block_size, block_size] * V:[block_size, S]
                :         //   aka: M:1~block_size, N:S, K:block_size
                :         // Because K and V are from cache, can use M2'=rnd_up(M2, block_size) to simplify logic
            1280 :         auto in_type = precision_of<DATA_TYPE>::value;
            1280 :         this->H = H;
            1280 :         this->S = S;
            1280 :         this->SV = SV;
            1280 :         this->Hk = Hk;
            1280 :         _h_each_group_len = h_each_group_len;
            1280 :         _block_size = block_size;
            1280 :         _nthr = static_cast<size_t>(parallel_get_max_threads());
            1280 :         _sliding_window = sliding_window;
            1280 :         _d_scale = d_scale;
                : #    if defined(OPENVINO_ARCH_ARM64)
            1280 :         AarchF16 = one_of(precision_of<DATA_TYPE>::value, ov::element::f16) ? true : false;
                : #    endif
            1280 :         auto prev_score_stride = _weight.stride(2);
            1280 :         auto want_score_stride = rnd_up(kv_len, _block_size);
            1280 :         auto new_score_stride = std::max(prev_score_stride, want_score_stride);
                :         // resize temporary buffers, weight.size(3) will be aligned to block_size
            2560 :         _weight.resize<float>({static_cast<size_t>(_nthr), H, _block_size, new_score_stride});
                :         // std::max(S, SV) here is to ensure by_channel quantize has enough buffer to use
            1280 :         if (_quant_key_bychannel)
               0 :             _output.resize<float>({static_cast<size_t>(_nthr), _block_size, H, std::max(S, SV)});
                :         else
            2560 :             _output.resize<float>({static_cast<size_t>(_nthr), _block_size, H, SV});
                :
                :         // TODO: kernel supports stride
            1280 :         if (!AarchF16 && (_qk_gemm.empty() || prev_score_stride < new_score_stride)) {
               0 :             _qk_gemm.resize(_block_size);
               0 :             _wv_gemm.resize(_block_size);
               0 :             _wv_gemm_acc.resize(_block_size);
               0 :             for (size_t i = 0; i < _block_size; i++) {
               0 :                 _qk_gemm[i] = std::make_shared<BrgemmKernel>(i + 1,
               0 :                                                              _block_size,
                :                                                              S,
               0 :                                                              H * S,
                :                                                              _block_size,
               0 :                                                              _weight.stride(2),
               0 :                                                              false,
                :                                                              in_type);
               0 :                 _wv_gemm[i] =
               0 :                     std::make_shared<BrgemmKernel>(i + 1,
                :                                                    SV,
                :                                                    _block_size,
                :                                                    // if it's bf16, the stride needs double due to reuse float buffer
               0 :                                                    (in_type == ov::element::Type_t::f32 ? 1 : 2) * _weight.stride(2),
```

```
1654           :                                                 SV,
1655         0 :                                                 _output.stride(1),
1656         0 :                                                 false,
1657           :                                                 in_type);
1658         0 :                 _wv_gemm_acc[i] =
1659         0 :                     std::make_shared<BrgemmKernel>(i + 1,
1660           :                                                 SV,
1661           :                                                 _block_size,
1662           :                                                 // if it's bf16, the stride needs double due to reuse float buffer
1663         0 :                                                 (in_type == ov::element::Type_t::f32 ? 1 : 2) * _weight.stride(2),
1664           :                                                 SV,
1665         0 :                                                 _output.stride(1),
1666         0 :                                                 false,
1667           :                                                 in_type,
1668         0 :                                                 true);
1669           :             }
1670           :
1671           :             // wsp is used to compute beta when K is blocked
1672         0 :             _wsp_size_per_thread = _wv_gemm[0]->get_wsp_size();
1673         0 :             _wsp.resize(_nthr * _wsp_size_per_thread);
1674           :
1675           :             // allocate scratch a/b, notice get_scratch_a_size/get_scratch_b_size returns in bytes
1676         0 :             _qk_scratch_a.resize<DATA_TYPE>(
1677         0 :                 {_nthr, _qk_gemm[_block_size - 1]->get_scratch_a_size() / sizeof(DATA_TYPE)});
1678         0 :             _wv_scratch_a.resize<DATA_TYPE>(
1679         0 :                 {_nthr, _wv_gemm[_block_size - 1]->get_scratch_a_size() / sizeof(DATA_TYPE)});
1680           :
1681           : #    if defined(OPENVINO_ARCH_X86_64)
1682           :             if ((S % 32 == 0) && (block_size % 16 == 0) && (S <= 32 * 6)) {
1683           :                 if (dnnl::impl::cpu::x64::mayiuse(dnnl::impl::cpu::x64::amx_bf16) &&
1684           :                     precision_of<DATA_TYPE>::value == ov::element::bf16 &&
1685           :                     precision_of<KEY_CACHE_TYPE>::value == ov::element::bf16 && VALUE_PREC == ov::element::bf16) {
1686           :                     _fastpath_valid_prec = ov::element::bf16;
1687           :                 } else if (dnnl::impl::cpu::x64::mayiuse(dnnl::impl::cpu::x64::amx_fp16) &&
1688           :                            precision_of<DATA_TYPE>::value == ov::element::f16 &&
1689           :                            precision_of<KEY_CACHE_TYPE>::value == ov::element::f16 && VALUE_PREC == ov::element::f16) {
1690           :                     _fastpath_valid_prec = ov::element::f16;
1691           :                 }
1692           :             }
1693           :             if (one_of(_fastpath_valid_prec, ov::element::bf16, ov::element::f16) && !_gemv) {
1694           :                 _gemv = std::make_shared<JitMatMulVecAMX>(static_cast<int>(S),
1695           :                                                           static_cast<int>(block_size),
1696           :                                                           _fastpath_valid_prec);
1697           :             }
1698           : #    endif
1699           :         }
1700           :
1701      1280 :         if (init_alibi_lookup && (!_alibi_lookup || _alibi_lookup.m_dims[0] < kv_len)) {
1702         0 :             _alibi_lookup.resize<float>({kv_len * 2});
1703         0 :             for (size_t i = 0; i < _alibi_lookup.m_dims[0]; i++) {
1704         0 :                 _alibi_lookup.ptr<float>()[i] = -static_cast<int>((_alibi_lookup.m_dims[0] - 1 - i));
1705           :             }
1706           :         }
1707           :
1708      1280 :         if (init_rotation_coefficient_scratch) {
1709         0 :             _block_rotation_coefficient_scratch.resize<DATA_TYPE>({_block_size, S});
1710           :         }
1711      1280 :     }
1712           :
1713        32 :     void init_reorder_buffers(size_t batch, size_t kv_len_in_blocks) {
1714        64 :         _qk_scratch_b.resize<DATA_TYPE>({batch, kv_len_in_blocks, Hk, _block_size * S});
1715        32 :         if (AarchF16) {
1716           :             // It is required to keep kv_cache continuous in mem, as kleidi do not support accumulation
1717        64 :             _wv_scratch_b.resize<DATA_TYPE>({batch, Hk, kv_len_in_blocks, _block_size * rnd_up(SV, _block_size)});
1718           :         } else {
1719         0 :             _wv_scratch_b.resize<DATA_TYPE>({batch, kv_len_in_blocks, Hk, _block_size * rnd_up(SV, _block_size)});
1720           :         }
1721        32 :     }
1722           :
1723         0 :     void init_score_buffers(const PlainTensor& past_lens, const PlainTensor& subsequence_begins) {
1724           :         static constexpr int cache_line_size = dnnl::impl::cpu::platform::get_cache_line_size();
1725         0 :         auto seq_cout = static_cast<int32_t>(past_lens.m_dims[0]);
1726         0 :         _score_offsets_aligned.resize<int32_t>({past_lens.m_dims[0]});
1727         0 :         _score_offsets.resize<int32_t>({past_lens.m_dims[0]});
1728         0 :         int32_t total_kv_len_aligned = 0;
1729         0 :         int32_t total_kv_len = 0;
1730         0 :         for (int32_t i = 0; i < seq_cout; i++) {
1731         0 :             auto q_len = subsequence_begins.ptr<int32_t>()[i + 1] - subsequence_begins.ptr<int32_t>()[i];
1732         0 :             auto kv_len = past_lens.ptr<int32_t>()[i] + q_len;
1733         0 :             _score_offsets_aligned.ptr<int32_t>()[i] = total_kv_len_aligned;
1734         0 :             _score_offsets.ptr<int32_t>()[i] = total_kv_len;
1735           :             // aligned to cache line to avoid false sharing
1736         0 :             total_kv_len_aligned += rnd_up(kv_len, cache_line_size / sizeof(float));
1737         0 :             total_kv_len += kv_len;
1738           :         }
1739           :
1740         0 :         _score_output.resize<float>({total_kv_len_aligned * H});
1741         0 :     }
1742           :
1743           :     // compute one block(such as 32 tokens) of query in M dimension: softmax(q_block*k')*v
1744           :     // all tensors such as query... have no batch dimension because batch dimension is varying
1745           :     //  query: [H, L, S]
1746           :     //  present_value: [block_number, H, 32, S]
1747           :     //  output_emb: [L, H * S]
1748           :     //  qk_scratch_b: [rnd_up(kv_len, block_size), Hk, scratch_b_size]
1749           :     //  wv_scratch_b: [rnd_up(kv_len, block_size), Hk, scratch_b_size]
1750         0 :     void exec_kernel_multiple(const PlainTensor& query,
1751           :                               const PlainTensor& present_value,
1752           :                               const PlainTensor& output_emb,
1753           :                               const PlainTensor& qk_scratch_b,
1754           :                               const PlainTensor& wv_scratch_b,
1755           :                               const int32_t* block_table,
1756           :                               size_t ithr,
1757           :                               size_t q_blk,
```

```
1758             :                                size_t hq_beg,
1759             :                                size_t hq_end,
1760             :                                size_t hk,
1761             :                                size_t q_len,
1762             :                                size_t cur_kv_len,
1763             :                                const PlainTensor& alibi_slopes,
1764             :                                float* score_output) {
1765           0 :         auto q_start = q_blk * _block_size;
1766           0 :         auto q_end = std::min(q_start + _block_size, q_len);
1767           0 :         auto q_cnt = q_end - q_start;
1768           0 :         constexpr bool q_is_xf16 = one_of(precision_of<DATA_TYPE>::value, ov::element::bf16, ov::element::f16);
1769           0 :         constexpr bool q_cache_is_same = precision_of<DATA_TYPE>::value == VALUE_PREC;
1770           0 :         auto cur_kv_len_blocks = div_up(cur_kv_len, _block_size);
1771           0 :         for (size_t h = hq_beg; h < hq_end; h++) {
1772           0 :             auto* q_ptr = query.ptr<DATA_TYPE>(h, q_start, 0);
1773           0 :             auto* c_ptr = _weight.ptr<float>(ithr, h, 0, 0);
1774             :             // for each query block, loop through all key block
1775             :             // for blocks:
1776             :             // 1 0 0 0 ...
1777             :             // 1 1 0 0 ...
1778             :             // 1 1 1 0 ...
1779             :             // just computing the positions of 1 should be enough
1780           0 :             for (size_t k_blk = 0; k_blk < cur_kv_len_blocks; k_blk++) {
1781           0 :                 auto* k_ptr = qk_scratch_b.ptr<DATA_TYPE>(k_blk, hk);
1782           0 :                 _qk_gemm[q_cnt - 1]->executeGemm(q_cnt < _block_size,
1783             :                                                  q_ptr,
1784             :                                                  k_ptr,
1785           0 :                                                  c_ptr + k_blk * _block_size,
1786           0 :                                                  _wsp.data() + ithr * _wsp_size_per_thread,
1787           0 :                                                  _qk_scratch_a ? _qk_scratch_a.ptr<DATA_TYPE>(ithr, 0) : nullptr);
1788             :             }
1789             :
1790           0 :             for (size_t m = q_start; m < q_end; m++) {
1791             :                 // apply attention mask & sofmax
1792           0 :                 auto ncausal = (cur_kv_len - q_cnt + (m - q_start) + 1);
1793           0 :                 auto score = _weight.ptr<float>(ithr, h, m - q_start);
1794           0 :                 if (_sliding_window) {
1795           0 :                     size_t start_idx = 0;
1796           0 :                     auto new_causal = ncausal;
1797           0 :                     float* alibi_lookup = nullptr;
1798           0 :                     if (ncausal > _sliding_window) {
1799           0 :                         start_idx = ncausal - static_cast<size_t>(_sliding_window);
1800           0 :                         new_causal = _sliding_window;
1801             :                     }
1802           0 :                     attn_softmax_kernel<float>(score + start_idx,
1803             :                                                reinterpret_cast<DATA_TYPE*>(score) + start_idx,
1804             :                                                _d_scale,
1805             :                                                alibi_lookup,
1806             :                                                nullptr,
1807             :                                                nullptr,
1808             :                                                false,
1809             :                                                new_causal,
1810           0 :                                                rnd_up(cur_kv_len, _block_size) - start_idx,
1811             :                                                precision_of<DATA_TYPE>::value,
1812             :                                                precision_of<DATA_TYPE>::value);
1813             :
1814           0 :                     memset(score, 0, sizeof(DATA_TYPE) * start_idx);
1815             :                 } else {
1816             :                     // alibi may available when _sliding_window is false
1817           0 :                     float* alibi_lookup = nullptr;
1818           0 :                     float alibi_slope = 0.f;
1819           0 :                     if (alibi_slopes) {
1820           0 :                         alibi_slope = alibi_slopes.ptr<float>()[h];
1821           0 :                         alibi_lookup = _alibi_lookup.ptr<float>() + _alibi_lookup.m_dims[0] - ncausal;
1822             :                     }
1823           0 :                     attn_softmax_kernel<float>(score,
1824             :                                                reinterpret_cast<DATA_TYPE*>(score),
1825             :                                                _d_scale,
1826             :                                                alibi_lookup,
1827             :                                                nullptr,
1828             :                                                nullptr,
1829             :                                                false,
1830             :                                                ncausal,
1831             :                                                rnd_up(cur_kv_len, _block_size),
1832             :                                                precision_of<DATA_TYPE>::value,
1833             :                                                precision_of<DATA_TYPE>::value,
1834             :                                                alibi_slope);
1835             :                 }
1836           0 :                 if (score_output) {
1837           0 :                     cvt_copy(score_output + h * rnd_up(cur_kv_len, 16),
1838             :                              reinterpret_cast<DATA_TYPE*>(score),
1839             :                              1,
1840             :                              cur_kv_len,
1841             :                              0,
1842             :                              0);
1843             :                 }
1844             :             }
1845             :
1846             :             // reuse float buffer, need to use float to compute offset
1847           0 :             auto* w_ptr = reinterpret_cast<DATA_TYPE*>(_weight.ptr<float>(ithr, h, 0, 0));
1848             :             float* fp32_out_ptr =
1849           0 :                 q_is_xf16 ? _output.ptr<float>(ithr, 0, h, 0) : output_emb.ptr<float>(q_start, h * SV);
1850             :
1851             :             // for each weight block, loop through all value block
1852           0 :             for (size_t v_blk = 0; v_blk < cur_kv_len_blocks; v_blk++) {
1853             :                 DATA_TYPE* v_ptr;
1854             :                 if (q_is_xf16 || !q_cache_is_same) {
1855           0 :                     v_ptr = wv_scratch_b.ptr<DATA_TYPE>(v_blk, hk);
1856             :                 } else {
1857             :                     v_ptr = present_value.ptr<DATA_TYPE>(block_table[v_blk], hk);
1858             :                 }
1859           0 :                 if (v_blk == 0) {
1860           0 :                     _wv_gemm[q_cnt - 1]->executeGemm(q_cnt < _block_size,
1861           0 :                                                      w_ptr + v_blk * _block_size,
```

```
1862           :                                                 v_ptr,
1863           :                                                 fp32_out_ptr,
1864         0 :                                                 _wsp.data() + ithr * _wsp_size_per_thread,
1865         0 :                                                 _wv_scratch_a ? _wv_scratch_a.ptr<DATA_TYPE>(ithr, 0) : nullptr);
1866           :                     } else {
1867         0 :                         _wv_gemm_acc[q_cnt - 1]->executeGemm(
1868         0 :                             q_cnt < _block_size,
1869         0 :                             w_ptr + v_blk * _block_size,
1870           :                             v_ptr,
1871           :                             fp32_out_ptr,
1872         0 :                             _wsp.data() + ithr * _wsp_size_per_thread,
1873         0 :                             _wv_scratch_a ? _wv_scratch_a.ptr<DATA_TYPE>(ithr, 0) : nullptr);
1874           :                     }
1875           :                 }
1876           :                 if (q_is_xf16) {
1877           :                     attn_memcpy2d_kernel(_output.ptr<float>(ithr, 0, h, 0),
1878           :                                          output_emb.ptr<DATA_TYPE>(q_start, h * SV),
1879           :                                          ov::element::f32,
1880           :                                          precision_of<DATA_TYPE>::value,
1881           :                                          _output.stride(1),
1882           :                                          output_emb.stride(0),
1883           :                                          SV,
1884           :                                          q_cnt);
1885           :                 }
1886           :             }
1887         0 :         }
1888           : #    if defined(OPENVINO_ARCH_ARM64)
1889           :     // compute one block(such as 32 tokens) of query in M dimension: softmax(q_block*k')*v
1890           :     // all tensors such as query... have no batch dimension because batch dimension is varying
1891           :     //  query: [H, L, S]
1892           :     //  present_value: [block_number, H, 32, S]
1893           :     //  output_emb: [L, H * S]
1894           :     //  qk_scratch_b: [rnd_up(kv_len, block_size), Hk, scratch_b_size]
1895           :     //  wv_scratch_b: [rnd_up(kv_len, block_size), Hk, scratch_b_size]
1896      3034 :     void exec_kernel_multiple_kai(const PlainTensor& query,
1897           :                                   const PlainTensor& present_value,
1898           :                                   const PlainTensor& output_emb,
1899           :                                   const PlainTensor& qk_scratch_b,
1900           :                                   const PlainTensor& wv_scratch_b,
1901           :                                   const int32_t* block_table,
1902           :                                   size_t ithr,
1903           :                                   size_t q_blk,
1904           :                                   size_t hq_beg,
1905           :                                   size_t hq_end,
1906           :                                   size_t hk,
1907           :                                   size_t q_len,
1908           :                                   size_t cur_kv_len,
1909           :                                   const PlainTensor& alibi_slopes,
1910           :                                   float* score_output) {
1911      3034 :         auto q_start = q_blk * _block_size;
1912      3034 :         auto q_end = std::min(q_start + _block_size, q_len);
1913      3034 :         auto q_cnt = q_end - q_start;
1914      3034 :         constexpr bool q_is_xf16 = one_of(precision_of<DATA_TYPE>::value, ov::element::bf16, ov::element::f16);
1915      3034 :         auto cur_kv_len_blocks = div_up(cur_kv_len, _block_size);
1916      2997 :         auto _score_stride = _weight.stride_bytes(2) / 2;
1917      6863 :         for (size_t h = hq_beg; h < hq_end; h++) {
1918      3012 :             auto* q_ptr = query.ptr<DATA_TYPE>(h, q_start, 0);
1919      3012 :             float* c_ptr = _weight.ptr<float>(ithr, h, 0, 0);
1920           :             // for each query block, loop through all key block
1921           :             // for blocks:
1922           :             // 1 0 0 0 ...
1923           :             // 1 1 0 0 ...
1924           :             // 1 1 1 0 ...
1925           :             // just computing the positions of 1 should be enough
1926      6421 :             for (size_t k_blk = 0; k_blk < cur_kv_len_blocks; k_blk++) {
1927      2908 :                 auto* k_ptr = qk_scratch_b.ptr<DATA_TYPE>(k_blk, hk);
1928      2908 :                 auto* qk_out_ptr =
1929           :                     c_ptr +
1930      2908 :                     (precision_of<DATA_TYPE>::value == ov::element::Type_t::f32 ? _block_size : _block_size / 2) *
1931           :                         k_blk;
1932           :
1933      2908 :                 KleidiKernel qkKernel(q_cnt, _block_size, S, H * S, _block_size, _score_stride);
1934      2868 :                 PlainTensor bias_k;
1935      6098 :                 bias_k.resize<float16_t>({_block_size});
1936      3673 :                 memset(bias_k.ptr<float16_t>(0), 0, sizeof(DATA_TYPE) * _block_size);
1937      3673 :                 PlainTensor packedB_k;
1938      6830 :                 packedB_k.resize<float16_t>({qkKernel.get_packed_rhs_size()});
1939      3638 :                 qkKernel.packB(reinterpret_cast<float16_t*>(k_ptr),
1940           :                                packedB_k.ptr<float16_t>(0),
1941           :                                bias_k.ptr<float16_t>(0));
1942      3869 :                 qkKernel.executeGemm(q_ptr, packedB_k.ptr<float16_t>(0), qk_out_ptr);
1943           :             }
1944           :
1945     79055 :             for (size_t m = q_start; m < q_end; m++) {
1946           :                 // apply softmax in f32 precision
1947     13926 :                 auto ncausal = (cur_kv_len - q_cnt + (m - q_start) + 1);
1948     13926 :                 auto soft_in = _weight.ptr<float>(ithr, h, m - q_start);
1949     13926 :                 auto score = _weight.ptr<float>(ithr, h, m - q_start);
1950     13926 :                 PlainTensor f32_cvt;
1951           :                 if (q_is_xf16) {
1952     79340 :                     f32_cvt.resize<float>({size_t{rnd_up(cur_kv_len, _block_size)}});
1953     74433 :                     cvt_copy(f32_cvt.ptr<float>(0),
1954           :                              reinterpret_cast<DATA_TYPE*>(score),
1955           :                              rnd_up(cur_kv_len, _block_size));
1956     76427 :                     soft_in = f32_cvt.ptr<float>(0);
1957           :                 }
1958     76427 :                 if (_sliding_window) {
1959         0 :                     size_t start_idx = 0;
1960         0 :                     auto new_causal = ncausal;
1961         0 :                     float* alibi_lookup = nullptr;
1962         0 :                     if (ncausal > _sliding_window) {
1963         0 :                         start_idx = ncausal - static_cast<size_t>(_sliding_window);
1964         0 :                         new_causal = _sliding_window;
1965           :                     }
```

```
1966          0 :                     attn_softmax_kernel<float>(soft_in + start_idx,
1967          0 :                                                reinterpret_cast<DATA_TYPE*>(score) + start_idx,
1968            :                                                _d_scale,
1969            :                                                alibi_lookup,
1970            :                                                nullptr,
1971            :                                                nullptr,
1972            :                                                false,
1973            :                                                new_causal,
1974          0 :                                                rnd_up(cur_kv_len, _block_size) - start_idx,
1975            :                                                precision_of<DATA_TYPE>::value,
1976            :                                                precision_of<DATA_TYPE>::value);
1977            :
1978          0 :                     memset(score, 0, sizeof(DATA_TYPE) * start_idx);
1979            :                 } else {
1980            :                     // alibi may available when _sliding_window is false
1981      76427 :                     float* alibi_lookup = nullptr;
1982      76427 :                     float alibi_slope = 0.f;
1983      76427 :                     if (alibi_slopes) {
1984          0 :                         alibi_slope = alibi_slopes.ptr<float>()[h];
1985          0 :                         alibi_lookup = _alibi_lookup.ptr<float>() + _alibi_lookup.m_dims[0] - ncausal;
1986            :                     }
1987      76427 :                     attn_softmax_kernel<float>(soft_in,
1988            :                                                reinterpret_cast<DATA_TYPE*>(score),
1989            :                                                _d_scale,
1990            :                                                alibi_lookup,
1991            :                                                nullptr,
1992            :                                                nullptr,
1993            :                                                false,
1994            :                                                ncausal,
1995            :                                                rnd_up(cur_kv_len, _block_size),
1996            :                                                precision_of<DATA_TYPE>::value,
1997            :                                                precision_of<DATA_TYPE>::value,
1998            :                                                alibi_slope);
1999            :                 }
2000      76540 :                 if (score_output) {
2001          0 :                     cvt_copy(score_output + h * rnd_up(cur_kv_len, 16),
2002            :                              reinterpret_cast<DATA_TYPE*>(score),
2003            :                              cur_kv_len);
2004            :                 }
2005            :             }
2006            :
2007            :             // reuse float buffer, need to use float to compute offset
2008       2612 :             auto* w_ptr = reinterpret_cast<DATA_TYPE*>(_weight.ptr<float>(ithr, h, 0, 0));
2009       2612 :             DATA_TYPE* out_ptr = output_emb.ptr<DATA_TYPE>(q_start, h * SV);
2010            :             DATA_TYPE* v_ptr;
2011       2612 :             v_ptr = wv_scratch_b.ptr<DATA_TYPE>(hk, 0);
2012       2612 :             PlainTensor bias;
2013       6526 :             bias.resize<float16_t>({SV});
2014       4010 :             memset(bias.ptr<float16_t>(0), 0, sizeof(DATA_TYPE) * SV);
2015       4010 :             PlainTensor packedB;
2016       4010 :             KleidiKernel wvKernel(q_cnt, SV, _block_size * cur_kv_len_blocks, _score_stride, SV, H * SV);
2017       7754 :             packedB.resize<float16_t>({wvKernel.get_packed_rhs_size()});
2018       4046 :             wvKernel.packB(reinterpret_cast<float16_t*>(v_ptr), packedB.ptr<float16_t>(0), bias.ptr<float16_t>(0));
2019       4043 :             wvKernel.executeGemm(reinterpret_cast<float16_t*>(w_ptr), packedB.ptr<float16_t>(0), out_ptr);
2020            :         }
2021       3183 :     }
2022            : #    endif
2023            :
2024            :     // compute one token, loop along batch and head dimensions
2025            :     // all tensors such as query... have no batch dimension because batch dimension is varying
2026            :     //  query: [H, L, S]
2027            :     //  present_*: [block_number, H, 32, S]
2028            :     //  output_emb: [L, H * S]
2029            :     //  weight: [nthr, H, 32, rnd_up(kv_len, block_size)]
2030            :     //  output: [nthr, 32, H, S]
2031          0 :     void exec_kernel_one_bh(const PlainTensor& query,
2032            :                             const PlainTensor& present_key,
2033            :                             const PlainTensor& present_value,
2034            :                             const PlainTensor& output_emb,
2035            :                             const int32_t* block_table,
2036            :                             size_t ithr,
2037            :                             size_t hq_beg,
2038            :                             size_t hq_end,
2039            :                             size_t hk,
2040            :                             size_t q_len,
2041            :                             size_t cur_kv_len,
2042            :                             const PlainTensor& alibi_slopes,
2043            :                             float* score_output) {
2044            : #    if defined(OPENVINO_ARCH_X86_64)
2045            :         if (one_of(_fastpath_valid_prec, ov::element::bf16, ov::element::f16)) {
2046            :             _gemv->tile_config();
2047            :             for (size_t pk = 0, i = 0; pk < cur_kv_len; pk += _block_size, i++) {
2048            :                 auto block_number = block_table[i];
2049            :                 for (size_t pq = 0; pq < q_len; pq++) {
2050            :                     for (size_t h = hq_beg; h < hq_end; h++) {
2051            :                         (*_gemv)(query.ptr<DATA_TYPE>(h, pq),
2052            :                                  present_key.ptr<KEY_CACHE_TYPE>(block_number, hk),
2053            :                                  _weight.ptr<float>(ithr, h, pq) + pk);
2054            :                     }
2055            :                 }
2056            :             }
2057            :             _gemv->tile_release();
2058            :         } else {
2059            : #    endif
2060          0 :             for (size_t pk = 0, i = 0; pk < cur_kv_len; pk += _block_size, i++) {
2061          0 :                 auto block_number = block_table[i];
2062          0 :                 for (size_t pq = 0; pq < q_len; pq++) {
2063          0 :                     for (size_t h = hq_beg; h < hq_end; h++) {
2064          0 :                         if (precision_of<KEY_CACHE_TYPE>::value == ov::element::u8 && _quant_key_bychannel) {
2065          0 :                             dot_product_block_by_channel(query.ptr<DATA_TYPE>(h, pq),
2066            :                                                          present_key.ptr<uint8_t>(block_number, hk),
2067          0 :                                                          _weight.ptr<float>(ithr, h, pq) + pk,
2068            :                                                          S,
2069          0 :                                                          std::min(_block_size, cur_kv_len - pk));
```

```
2070             :                         } else {
2071           0 :                             dot_product_block(query.ptr<DATA_TYPE>(h, pq),
2072             :                                               present_key.ptr<KEY_CACHE_TYPE>(block_number, hk),
2073           0 :                                               _weight.ptr<float>(ithr, h, pq) + pk,
2074             :                                               S,
2075           0 :                                               std::min(_block_size, cur_kv_len - pk),
2076             :                                               _key_group_size);
2077             :                         }
2078             :                     }
2079             :                 }
2080             :             }
2081             : #    if defined(OPENVINO_ARCH_X86_64)
2082             :         }
2083             : #    endif
2084             :
2085           0 :         for (size_t pq = 0; pq < q_len; pq++) {
2086           0 :             for (size_t h = hq_beg; h < hq_end; h++) {
2087             :                 // apply attention mask & sofmax
2088           0 :                 float* alibi_lookup = nullptr;
2089           0 :                 float alibi_slope = 0.f;
2090           0 :                 if (alibi_slopes) {
2091           0 :                     alibi_slope = alibi_slopes.ptr<float>()[h];
2092           0 :                     alibi_lookup = _alibi_lookup.ptr<float>() + _alibi_lookup.m_dims[0] - cur_kv_len;
2093             :                 }
2094           0 :                 attn_softmax_kernel<float>(_weight.ptr<float>(ithr, h, pq),
2095           0 :                                            _weight.ptr<float>(ithr, h, pq),
2096             :                                            _d_scale,
2097             :                                            alibi_lookup,
2098             :                                            nullptr,
2099             :                                            nullptr,
2100             :                                            false,
2101             :                                            cur_kv_len,
2102             :                                            cur_kv_len,
2103             :                                            ov::element::f32,
2104             :                                            ov::element::f32,
2105             :                                            alibi_slope);
2106           0 :                 if (score_output) {
2107           0 :                     memcpy(score_output + h * rnd_up(cur_kv_len, 16),
2108           0 :                            _weight.ptr<float>(ithr, h, pq),
2109             :                            cur_kv_len * sizeof(float));
2110             :                 }
2111             :             }
2112             :         }
2113             :
2114           0 :         memset(_output.ptr<float>(ithr), 0, q_len * H * SV * sizeof(float));
2115           0 :         for (size_t pv = 0, i = 0; pv < cur_kv_len; pv += _block_size, i++) {
2116           0 :             auto block_number = block_table[i];
2117           0 :             for (size_t pq = 0; pq < q_len; pq++) {
2118           0 :                 for (size_t h = hq_beg; h < hq_end; h++) {
2119           0 :                     auto sub_byte_multiplier = get_sub_byte_multiplier(present_value.get_precision());
2120           0 :                     size_t v_stride = (block_number * present_value.m_strides[0] + hk * present_value.m_strides[1]) *
2121           0 :                                       present_value.get_precision().size() / sub_byte_multiplier;
2122           0 :                     auto* v_ptr = reinterpret_cast<typename element_type_traits<VALUE_PREC>::value_type*>(
2123           0 :                         present_value.m_ptr.get() + v_stride);
2124           0 :                     attn_acc_value_block<typename element_type_traits<VALUE_PREC>::value_type, VALUE_PREC>(
2125             :                         _output.ptr<float>(ithr, pq, h),
2126           0 :                         _weight.ptr<float>(ithr, h, pq) + pv,
2127             :                         v_ptr,
2128             :                         SV,
2129           0 :                         std::min(_block_size, cur_kv_len - pv),
2130             :                         _value_group_size);
2131             :                 }
2132             :             }
2133             :         }
2134             :         // convert to dst
2135           0 :         for (size_t pq = 0; pq < q_len; pq++) {
2136           0 :             for (size_t h = hq_beg; h < hq_end; h++) {
2137           0 :                 cvt_copy(output_emb.ptr<DATA_TYPE>(pq, h * SV), _output.ptr<float>(ithr, pq, h), 1, SV, 0, 0);
2138             :             }
2139             :         }
2140           0 :     }
2141             :
2142             :     // compute one token, loop along batch, head dimensions and kv_len, it's special for very long kv_len with small
2143             :     // batch tokens. It will assume NO mixture execution of first and second token. all tensors such as query... have
2144             :     // batch dimension which is DIFFERENT from above
2145             :     //  query: [B, H, L, S]
2146             :     //  key_cache: [block_number, H, _block_size, S]
2147             :     //  value_cache: [block_number, H, _block_size, Sv]
2148             :     //  output_emb: [B, L, H * S]
2149             :     // 3 loops along batch, head, kv cache length dimensions
2150        1248 :     void exec_loop_bhl(const PlainTensor& query,
2151             :                        PlainTensor& key_cache,
2152             :                        PlainTensor& value_cache,
2153             :                        const PlainTensor& output_emb,
2154             :                        const PlainTensor& output_score,
2155             :                        size_t max_context_len,
2156             :                        const PlainTensor& past_lens,
2157             :                        const PlainTensor& subsequence_begins,
2158             :                        const PlainTensor& block_indices,
2159             :                        const PlainTensor& block_indices_begins,
2160             :                        const PlainTensor& alibi_slopes) {
2161        1248 :         auto B = past_lens.size(0);
2162        1248 :         auto q_len = query.size(2);
2163        1248 :         auto kv_len_in_blocks = div_up(max_context_len, _block_size);
2164             :         // aligned to cache line (64bytes=16*sizeof(float)) to avoid false sharing
2165        2496 :         _weight_bhl.resize<float>({B, H, q_len, rnd_up(max_context_len, std::max(_block_size, size_t{16}))});
2166             :
2167             :         // for small batches dynamic scheduler has notable overhead
2168             :         bool prefer_static_loop;
2169             :         // if less than 2 work items per thread, loop H
2170        1248 :         bool loop_hk = B * kv_len_in_blocks * Hk <= 2 * _nthr ? false : true;
2171        1248 :         if (B <= 32) {
2172        1248 :             prefer_static_loop = true;
2173             :             // small batch and all batch size is same(like SDPA case)
```

```
2174        1248 :             auto kv_len = past_lens.ptr<int32_t>()[0];
2175        4992 :             for (size_t b = 1; b < B; b++) {
2176        3744 :                 if (past_lens.ptr<int32_t>()[b] != kv_len) {
2177        3744 :                     prefer_static_loop = false;
2178             :                 }
2179             :             }
2180             :         } else {
2181             :             // for bigger batch skip the test to save the cost
2182             :             prefer_static_loop = false;
2183             :         }
2184             :         auto get_h_params =
2185      487341 :             [](bool loop_hk, size_t hx, size_t h_each_group_len, size_t& hq_beg, size_t& hq_end, size_t& hk) {
2186      487341 :                 if (loop_hk) {
2187      465058 :                     hk = hx;
2188      465058 :                     hq_beg = hk * h_each_group_len;
2189      465058 :                     hq_end = (hk + 1) * h_each_group_len;
2190             :                 } else {
2191       22283 :                     hq_beg = hx;
2192       22283 :                     hq_end = hx + 1;
2193       22283 :                     hk = hx / h_each_group_len;
2194             :                 }
2195             :             };
2196      660062 :         auto loop_qk = [&](size_t b, size_t pk_in_blocks, size_t hx) {
2197      228494 :             auto context_len = static_cast<size_t>(past_lens.ptr<int32_t>()[b]) + 1;
2198             :             size_t hk, hq_beg, hq_end;
2199      228494 :             get_h_params(loop_hk, hx, _h_each_group_len, hq_beg, hq_end, hk);
2200             :
2201             :             // kv_len must be valid
2202      228494 :             auto pk = pk_in_blocks * _block_size;
2203      228494 :             if (pk < context_len) {
2204      214628 :                 auto block_number = block_indices.ptr<int32_t>()[block_indices_begins.ptr<int32_t>()[b] + pk_in_blocks];
2205             : #    if defined(OPENVINO_ARCH_X86_64)
2206             :                 if (one_of(_fastpath_valid_prec, ov::element::bf16, ov::element::f16)) {
2207             :                     _gemv->tile_config();
2208             :                     for (size_t pq = 0; pq < q_len; pq++) {
2209             :                         for (size_t h = hq_beg; h < hq_end; h++) {
2210             :                             (*_gemv)(query.ptr<DATA_TYPE>(b, h, pq),
2211             :                                      key_cache.ptr<KEY_CACHE_TYPE>(block_number, hk),
2212             :                                      _weight_bhl.ptr<float>(b, h, pq) + pk);
2213             :                         }
2214             :                     }
2215             :                     _gemv->tile_release();
2216             :                 } else {
2217             : #    endif
2218      316345 :                     for (size_t pq = 0; pq < q_len; pq++) {
2219      317409 :                         for (size_t h = hq_beg; h < hq_end; h++) {
2220      215692 :                             if (precision_of<KEY_CACHE_TYPE>::value == ov::element::u8 && _quant_key_bychannel) {
2221           0 :                                 dot_product_block_by_channel(query.ptr<DATA_TYPE>(b, h, pq),
2222             :                                                              key_cache.ptr<uint8_t>(block_number, hk),
2223           0 :                                                              _weight_bhl.ptr<float>(b, h, pq) + pk,
2224             :                                                              S,
2225           0 :                                                              std::min(_block_size, context_len - pk));
2226             :                             } else {
2227      431384 :                                 dot_product_block(query.ptr<DATA_TYPE>(b, h, pq),
2228             :                                                   key_cache.ptr<KEY_CACHE_TYPE>(block_number, hk),
2229      215692 :                                                   _weight_bhl.ptr<float>(b, h, pq) + pk,
2230             :                                                   S,
2231      344883 :                                                   std::min(_block_size, context_len - pk),
2232             :                                                   _key_group_size);
2233             :                             }
2234             :                         }
2235             :                     }
2236             : #    if defined(OPENVINO_ARCH_X86_64)
2237             :                 }
2238             : #    endif
2239             :             }
2240             :         };
2241             :
2242       60282 :         auto loop_softmax = [&](size_t b, size_t h, size_t pq) {
2243       59034 :             auto cur_kv_len = static_cast<size_t>(past_lens.ptr<int32_t>()[b]) + 1;
2244       59034 :             auto ncausal = cur_kv_len;
2245             :             // apply attention mask & sofmax
2246       59034 :             float* alibi_lookup = nullptr;
2247       59034 :             float alibi_slope = 0.f;
2248       59034 :             if (alibi_slopes) {
2249           0 :                 alibi_slope = alibi_slopes.ptr<float>()[h];
2250           0 :                 alibi_lookup = _alibi_lookup.ptr<float>() + _alibi_lookup.m_dims[0] - cur_kv_len;
2251             :             }
2252       59034 :             attn_softmax_kernel<float>(_weight_bhl.ptr<float>(b, h, pq),
2253       59034 :                                        _weight_bhl.ptr<float>(b, h, pq),
2254             :                                        _d_scale,
2255             :                                        alibi_lookup,
2256             :                                        nullptr,
2257             :                                        nullptr,
2258             :                                        false,
2259             :                                        ncausal,
2260             :                                        cur_kv_len,
2261             :                                        ov::element::f32,
2262             :                                        ov::element::f32,
2263             :                                        alibi_slope);
2264             :         };
2265             :
2266        1248 :         size_t h_dims = loop_hk ? Hk : H;
2267        1248 :         if (prefer_static_loop) {
2268           0 :             parallel_for3d(B, kv_len_in_blocks, h_dims, loop_qk);
2269           0 :             parallel_for3d(B, H, q_len, loop_softmax);
2270             :         } else {
2271        1248 :             parallel_for3d_dynamic(B, kv_len_in_blocks, h_dims, loop_qk);
2272        1248 :             parallel_for3d_dynamic(B, H, q_len, loop_softmax);
2273             :         }
2274             :
2275        1248 :         if (output_score) {
2276           0 :             parallel_for2d_dynamic(B, q_len, [&](size_t b, size_t pq) {
2277           0 :                 auto cur_kv_len = static_cast<size_t>(past_lens.ptr<int32_t>()[b]) + 1;
```

```
2278          0 :                 auto* src = _weight_bhl.ptr<float>(b, 0, pq);
2279          0 :                 size_t src_stride = _weight_bhl.stride(2);
2280          0 :                 auto* dst = output_score.ptr<float>() + _score_offsets.ptr<int32_t>()[b];
2281          0 :                 attn_reduce(dst, src, H, cur_kv_len, src_stride);
2282            :             });
2283            :         }
2284            :
2285            :         // attn_w * V
2286       2496 :         _output_bhl.resize<float>({static_cast<size_t>(_nthr), B, q_len, H, SV});
2287            :         // m_attn_w {B, H, q_len, kv_len}
2288      46855 :         parallel_nt_static(_nthr, [&](const size_t ithr, const size_t nthr) {
2289      22703 :             memset(_output_bhl.ptr<float>(ithr, 0, 0, 0, 0), 0, _output_bhl.stride(0) * sizeof(float));
2290            :         });
2291            :
2292     737940 :         auto loop_wk = [&](size_t b, size_t pv_in_blocks, size_t hx) {
2293     258847 :             auto ithr = parallel_get_thread_num();
2294     258847 :             auto context_len = static_cast<size_t>(past_lens.ptr<int32_t>()[b]) + 1;
2295     258847 :             auto pv = pv_in_blocks * _block_size;
2296            :             size_t hk, hq_beg, hq_end;
2297     258847 :             get_h_params(loop_hk, hx, _h_each_group_len, hq_beg, hq_end, hk);
2298            :
2299            :             // kv_len must be valid
2300     258847 :             if (pv < context_len) {
2301     241710 :                 auto block_number = block_indices.ptr<int32_t>()[block_indices_begins.ptr<int32_t>()[b] + pv_in_blocks];
2302     374157 :                 for (size_t pq = 0; pq < q_len; pq++) {
2303     373141 :                     for (size_t h = hq_beg; h < hq_end; h++) {
2304     240694 :                         auto sub_byte_multiplier = get_sub_byte_multiplier(value_cache.get_precision());
2305     242591 :                         size_t v_stride = (block_number * value_cache.m_strides[0] + hk * value_cache.m_strides[1]) *
2306     242591 :                                           value_cache.get_precision().size() / sub_byte_multiplier;
2307     255344 :                         auto* v_ptr = reinterpret_cast<typename element_type_traits<VALUE_PREC>::value_type*>(
2308     255344 :                             value_cache.m_ptr.get() + v_stride);
2309     255344 :                         attn_acc_value_block<typename element_type_traits<VALUE_PREC>::value_type, VALUE_PREC>(
2310            :                             _output_bhl.ptr<float>(ithr, b, pq, h),
2311     255344 :                             _weight_bhl.ptr<float>(b, h, pq) + pv,
2312            :                             v_ptr,
2313            :                             SV,
2314     399010 :                             std::min(_block_size, context_len - pv),
2315            :                             _value_group_size);
2316            :                     }
2317            :                 }
2318            :             }
2319            :         };
2320            :
2321       1248 :         if (prefer_static_loop) {
2322          0 :             parallel_for3d(B, kv_len_in_blocks, loop_hk ? Hk : H, loop_wk);
2323            :         } else {
2324       1248 :             parallel_for3d_dynamic(B, kv_len_in_blocks, loop_hk ? Hk : H, loop_wk);
2325            :         }
2326            :
2327     192303 :         parallel_for3d(B, H, q_len, [&](size_t b, size_t h, size_t pq) {
2328      59246 :             auto* temp = _output_bhl.ptr<float>(0, b, pq, h);
2329      59246 :             size_t temp_stride = _output_bhl.stride(0);
2330      56802 :             auto* dst = output_emb.ptr<DATA_TYPE>(b, pq, h * SV);
2331      56802 :             attn_reduce(dst, temp, _nthr, SV, temp_stride);
2332            :         });
2333       1248 :     }
2334            : };
2335            :
2336            : template <typename DATA_TYPE, typename KEY_CACHE_TYPE, ov::element::Type_t VALUE_PREC>
2337            : struct MHA {
2338            :     MHAHelper<DATA_TYPE, KEY_CACHE_TYPE, VALUE_PREC>& _helper;
2339            :     struct AttnWorkItem {
2340            :         int32_t batch_in_reorder;  // which batch in reorder buffer will be used
2341            :         int32_t batch_in_seq;      // batch idx in sequence
2342            :         int32_t q_len;             // current sequence length, 1 for second token, 2+ for first token
2343            :         int32_t q_block_id;        // block id in this seq, valid at first token
2344            :     };
2345            :     struct ReorderWorkItem {
2346            :         int32_t batch_in_seq;      // batch idx in sequence
2347            :         int32_t batch_in_reorder;  // which batch in reorder buffer will be used
2348            :         int32_t kv_block_id;       // block id in this kv cache seq
2349            :     };
2350         32 :     struct WorkItems {
2351            :     private:
2352            :         std::vector<AttnWorkItem> attn_items;
2353            :         std::vector<ReorderWorkItem> reorder_items;
2354            :         int32_t max_kv_len_in_reorder;  // max kv len between first tokens
2355            :         int32_t max_batch_in_reorder;
2356            :         int32_t total_kv_len;
2357            :
2358            :     public:
2359       1280 :         void reset(const PlainTensor& query,
2360            :                    const PlainTensor& past_lens,
2361            :                    const PlainTensor& subsequence_begins,
2362            :                    size_t block_size) {
2363       1280 :             attn_items.clear();
2364       1280 :             reorder_items.clear();
2365       1280 :             max_kv_len_in_reorder = 0;
2366       1280 :             max_batch_in_reorder = 0;
2367       1280 :             total_kv_len = 0;
2368            :
2369       1280 :             auto seq_cout = static_cast<int32_t>(past_lens.m_dims[0]);
2370       6400 :             for (int32_t i = 0; i < seq_cout; i++) {
2371       5120 :                 auto q_len = subsequence_begins.ptr<int32_t>()[i + 1] - subsequence_begins.ptr<int32_t>()[i];
2372       5120 :                 auto kv_len = past_lens.ptr<int32_t>()[i] + q_len;
2373       5120 :                 auto kv_len_in_block = static_cast<int32_t>(div_up(kv_len, block_size));
2374       5120 :                 if (q_len == 1) {
2375       4992 :                     attn_items.emplace_back(AttnWorkItem{0,     // batch_in_reorder
2376            :                                                          i,     // batch_in_seq
2377            :                                                          1ull,  // q_len
2378            :                                                          // kv_len in blocks, used in the sort function
2379       4992 :                                                          kv_len_in_block - 1});
2380            :                 } else {
2381        128 :                     auto reorder_sub_work_count = kv_len_in_block;
```

```
2382        128 :                     max_kv_len_in_reorder = std::max(max_kv_len_in_reorder, kv_len);
2383        256 :                     for (int32_t block_id = 0; block_id < reorder_sub_work_count; block_id++) {
2384        128 :                         reorder_items.emplace_back(ReorderWorkItem{
2385            :                             i,                      // batch_in_seq
2386        128 :                             max_batch_in_reorder,   // batch_in_reorder
2387            :                             block_id                // kv_block_id
2388            :                         });
2389            :                     }
2390            :
2391            :                     // workitems for attention
2392        128 :                     auto attn_sub_work_count = static_cast<int32_t>(div_up(q_len, block_size));
2393        256 :                     for (int32_t block_id = 0; block_id < attn_sub_work_count; block_id++) {
2394        128 :                         attn_items.emplace_back(AttnWorkItem{
2395        128 :                             max_batch_in_reorder,   // batch_in_reorder
2396            :                             i,                      // batch_in_seq
2397            :                             q_len,                  // q_len
2398            :                             block_id                // q_block_id
2399            :                         });
2400            :                     }
2401        128 :                     max_batch_in_reorder++;
2402            :                 }
2403       5120 :                 total_kv_len += kv_len;
2404            :             }
2405            :             // std::sort(attn_items.begin(), attn_items.end(), [] (const AttnWorkItem& left, const AttnWorkItem& right)
2406            :             // {
2407            :             //     // kv block number which will be acessed later
2408            :             //     auto left_kv_blocks = left.q_block_id;
2409            :             //     auto right_kv_blocks = right.q_block_id;
2410            :             //     return left_kv_blocks > right_kv_blocks;
2411            :             // });
2412       1280 :         }
2413       2811 :         [[nodiscard]] const AttnWorkItem& get_attn_work_item(size_t idx) const {
2414       2811 :             return attn_items[idx];
2415            :         }
2416         32 :         [[nodiscard]] size_t attn_work_size() const {
2417         32 :             return attn_items.size();
2418            :         }
2419       3021 :         [[nodiscard]] const ReorderWorkItem& get_reorder_work_item(size_t idx) const {
2420       3021 :             return reorder_items[idx];
2421            :         }
2422         32 :         [[nodiscard]] size_t reorder_work_size() const {
2423         32 :             return reorder_items.size();
2424            :         }
2425       1344 :         [[nodiscard]] size_t get_reorder_max_batch_size() const {
2426       1344 :             return static_cast<size_t>(max_batch_in_reorder);
2427            :         }
2428         32 :         [[nodiscard]] size_t get_reorder_max_kv_len() const {
2429         32 :             return static_cast<size_t>(max_kv_len_in_reorder);
2430            :         }
2431            :         [[nodiscard]] size_t get_total_kv_len() const {
2432            :             return static_cast<size_t>(total_kv_len);
2433            :         }
2434            :     };
2435            :
2436            :     WorkItems _workitems;
2437            :
2438         32 :     MHA(MHAHelper<DATA_TYPE, KEY_CACHE_TYPE, VALUE_PREC>& helper) : _helper(helper) {}
2439            :
2440            :     // one loop to handle first and second tokens
2441         32 :     void exec_loop_mixed(const PlainTensor& q,
2442            :                          PlainTensor& k_cache,
2443            :                          const PlainTensor& v_cache,
2444            :                          const PlainTensor& output_emb,
2445            :                          const PlainTensor& output_score,
2446            :                          size_t max_context_len,
2447            :                          const PlainTensor& past_lens,
2448            :                          const PlainTensor& subsequence_begins,
2449            :                          const PlainTensor& block_indices,
2450            :                          const PlainTensor& block_indices_begins,
2451            :                          const PlainTensor& alibi_slopes) {
2452         32 :         auto Hk = v_cache.m_dims[1];
2453            :
2454         32 :         constexpr bool q_is_xf16 = one_of(precision_of<DATA_TYPE>::value, ov::element::bf16, ov::element::f16);
2455         32 :         constexpr bool q_cache_is_same = precision_of<DATA_TYPE>::value == VALUE_PREC;
2456         32 :         auto attn_work_count = _workitems.attn_work_size();
2457         32 :         auto reorder_work_count = _workitems.reorder_work_size();
2458            :
2459            :         // buffer for transpose and repack
2460         32 :         _helper.init_reorder_buffers(_workitems.get_reorder_max_batch_size(),
2461            :                                      div_up(_workitems.get_reorder_max_kv_len(), _helper._block_size));
2462            :
2463            :         // packed k, v
2464       4060 :         parallel_for2d_dynamic(reorder_work_count, Hk, [&](size_t w, size_t hk) {
2465       3021 :             const auto& item = _workitems.get_reorder_work_item(w);
2466       3021 :             const auto batch_in_seq = item.batch_in_seq;
2467       3021 :             const auto batch_in_reorder = item.batch_in_reorder;
2468       3021 :             const auto kv_block = item.kv_block_id;
2469       3021 :             auto block_number =
2470       3021 :                 block_indices.ptr<int32_t>()[block_indices_begins.ptr<int32_t>()[batch_in_seq] + kv_block];
2471       3021 :             if (block_number < 0) {
2472            :                 return;
2473            :             }
2474            :
2475       3477 :             auto ithr = parallel_get_thread_num();
2476       3477 :             auto* k_ptr = k_cache.ptr<KEY_CACHE_TYPE>(block_number, hk);
2477            :
2478       3477 :             transpose_16NxK<DATA_TYPE, precision_of<KEY_CACHE_TYPE>::value>(
2479       3477 :                 _helper._qk_scratch_b.template ptr<DATA_TYPE>(batch_in_reorder, kv_block, hk),
2480            :                 k_ptr,
2481            :                 _helper._output.template ptr<DATA_TYPE>(ithr),
2482            :                 _helper._block_size,            // N
2483            :                 _helper.S,                      // K
2484            :                 _helper._block_size,            // dst_stride
2485            :                 _helper.S,                      // src_stride
```

```
2486             :                 _helper._key_group_size,        // group_size
2487             :                 _helper._quant_key_bychannel);  // quant_by_channel
2488             :
2489             :             if (q_is_xf16) {
2490        4028 :                 auto sub_byte_multiplier = get_sub_byte_multiplier(v_cache.get_precision());
2491        4026 :                 size_t v_stride = (block_number * v_cache.m_strides[0] + hk * v_cache.m_strides[1]) *
2492        4026 :                                   v_cache.get_precision().size() / sub_byte_multiplier;
2493        4038 :                 auto* v_ptr = v_cache.m_ptr.get() + v_stride;
2494             : #    if defined(OPENVINO_ARCH_ARM64)
2495        4038 :                 dequant<DATA_TYPE, VALUE_PREC>(
2496        4038 :                     _helper._wv_scratch_b.template ptr<DATA_TYPE>(batch_in_reorder, hk, kv_block),
2497             :                     v_ptr,
2498             :                     _helper._block_size,
2499             :                     _helper.SV,
2500             :                     _helper._value_group_size);
2501             : #    else
2502             :                 pack_32NxK<DATA_TYPE, VALUE_PREC>(
2503             :                     _helper._wv_scratch_b.template ptr<DATA_TYPE>(batch_in_reorder, kv_block, hk),
2504             :                     v_ptr,
2505             :                     _helper._output.template ptr<DATA_TYPE>(ithr),
2506             :                     _helper._block_size,
2507             :                     _helper.SV,
2508             :                     rnd_up(_helper.SV, _helper._block_size),
2509             :                     _helper.SV,
2510             :                     _helper._value_group_size);
2511             : #    endif
2512             :             } else {
2513             :                 // need to decompress
2514             :                 if (!q_cache_is_same) {
2515           0 :                     auto sub_byte_multiplier = get_sub_byte_multiplier(v_cache.get_precision());
2516           0 :                     size_t v_stride = (block_number * v_cache.m_strides[0] + hk * v_cache.m_strides[1]) *
2517           0 :                                       v_cache.get_precision().size() / sub_byte_multiplier;
2518           0 :                     auto* v_ptr = v_cache.m_ptr.get() + v_stride;
2519           0 :                     dequant<DATA_TYPE, VALUE_PREC>(
2520           0 :                         _helper._wv_scratch_b.template ptr<DATA_TYPE>(batch_in_reorder, kv_block, hk),
2521             :                         v_ptr,
2522             :                         _helper._block_size,
2523             :                         _helper.SV,
2524             :                         _helper._value_group_size);
2525             :                 }
2526             :             }
2527             :         });
2528             :
2529             :         // loop along HK dimension: if mixed first/second token and elements count is enough, loop HK to reuse KV in the
2530             :         // CPU cache
2531             :         //    else if elements count is small, prefer to loop H to get more work to avoid thread imbalance
2532          64 :         bool loop_hk = _workitems.get_reorder_max_batch_size() == past_lens.m_dims[0] ||  // if only first token, loop H
2533           0 :                                attn_work_count * Hk <= 2 * _helper._nthr
2534          32 :                            ? false
2535             :                            : true;  // or less than 2 work items per thread, loop H
2536             :
2537        9839 :         parallel_for2d_dynamic(attn_work_count, loop_hk ? Hk : _helper.H, [&](size_t w, size_t hx) {
2538             :             size_t hk, hq_beg, hq_end;
2539        2811 :             if (loop_hk) {
2540           0 :                 hk = hx;
2541           0 :                 hq_beg = hk * _helper._h_each_group_len;
2542           0 :                 hq_end = (hk + 1) * _helper._h_each_group_len;
2543             :             } else {
2544        2811 :                 hq_beg = hx;
2545        2811 :                 hq_end = hx + 1;
2546        2811 :                 hk = hx / _helper._h_each_group_len;
2547             :             }
2548             :
2549        2811 :             const auto& item = _workitems.get_attn_work_item(w);
2550        2811 :             const auto batch_in_seq = item.batch_in_seq;
2551        2811 :             const auto batch_in_token = subsequence_begins.ptr<int32_t>()[batch_in_seq];
2552        2811 :             const auto q_len = static_cast<size_t>(item.q_len);
2553        2894 :             size_t ithr = parallel_get_thread_num();
2554             :
2555        2894 :             if (q_len == 1) {
2556           0 :                 const auto cur_kv_len = static_cast<size_t>(past_lens.ptr<int32_t>()[batch_in_seq]) + 1;
2557           0 :                 float* score_output = nullptr;
2558           0 :                 if (output_score) {
2559           0 :                     auto score_offset = _helper._score_offsets_aligned.template ptr<int32_t>()[batch_in_seq];
2560           0 :                     score_output = _helper._score_output.template ptr<float>() + score_offset * _helper.H;
2561             :                 }
2562             :
2563           0 :                 _helper.exec_kernel_one_bh(
2564             :                     q.slice(0, batch_in_token, batch_in_token),
2565             :                     k_cache,
2566             :                     v_cache,
2567             :                     output_emb.slice(0, batch_in_token, batch_in_token),
2568           0 :                     block_indices.ptr<int32_t>() + block_indices_begins.ptr<int32_t>()[batch_in_seq],
2569             :                     ithr,
2570             :                     hq_beg,
2571             :                     hq_end,
2572             :                     hk,
2573             :                     1ul,
2574             :                     cur_kv_len,
2575             :                     alibi_slopes,
2576             :                     score_output);
2577             :             } else {
2578        2894 :                 const auto batch_in_reorder = item.batch_in_reorder;
2579        2894 :                 const auto q_blk = item.q_block_id;
2580        6037 :                 const auto q_cnt = std::min(_helper._block_size, q_len - q_blk * _helper._block_size);
2581        2894 :                 const auto cur_kv_len =
2582        2894 :                     static_cast<size_t>(past_lens.ptr<int32_t>()[batch_in_seq]) + q_blk * _helper._block_size + q_cnt;
2583        2894 :                 float* score_output = nullptr;
2584        2894 :                 if (output_score) {
2585             :                     // last block
2586           0 :                     if (q_len - q_blk * _helper._block_size <= _helper._block_size) {
2587           0 :                         auto score_offset = _helper._score_offsets_aligned.template ptr<int32_t>()[batch_in_seq];
2588           0 :                         score_output = _helper._score_output.template ptr<float>() + score_offset * _helper.H;
2589             :                     }
```

```
2590            :                 }
2591            :
2592         2894 :                 PlainTensor sub_query;
2593         5930 :                 sub_query.resize({q_len, _helper.H, _helper.S}, q.ptr<DATA_TYPE>(batch_in_token));
2594         3068 :                 sub_query = sub_query.permute({1, 0, 2});
2595            : #    if defined(OPENVINO_ARCH_ARM64)
2596            :                 if constexpr (q_is_xf16) {
2597         6112 :                     _helper.exec_kernel_multiple_kai(
2598            :                         sub_query,
2599            :                         v_cache,
2600         3236 :                         output_emb.slice(0, batch_in_token, batch_in_token + q_len)
2601         2927 :                             .reshape({q_len, _helper.H * _helper.SV}),
2602         2991 :                         _helper._qk_scratch_b.slice(0, batch_in_reorder, batch_in_reorder),
2603         3008 :                         _helper._wv_scratch_b.slice(0, batch_in_reorder, batch_in_reorder),
2604         3025 :                         block_indices.ptr<int32_t>() + block_indices_begins.ptr<int32_t>()[batch_in_seq],
2605            :                         ithr,
2606            :                         q_blk,
2607            :                         hq_beg,
2608            :                         hq_end,
2609            :                         hk,
2610            :                         q_len,
2611            :                         cur_kv_len,
2612            :                         alibi_slopes,
2613            :                         score_output);
2614            :                 } else {
2615            :                     _helper.exec_kernel_multiple(
2616            :                         sub_query,
2617            :                         v_cache,
2618            :                         output_emb.slice(0, batch_in_token, batch_in_token + q_len)
2619            :                             .reshape({q_len, _helper.H * _helper.SV}),
2620            :                         _helper._qk_scratch_b.slice(0, batch_in_reorder, batch_in_reorder),
2621            :                         _helper._wv_scratch_b.slice(0, batch_in_reorder, batch_in_reorder),
2622            :                         block_indices.ptr<int32_t>() + block_indices_begins.ptr<int32_t>()[batch_in_seq],
2623            :                         ithr,
2624            :                         q_blk,
2625            :                         hq_beg,
2626            :                         hq_end,
2627            :                         hk,
2628            :                         q_len,
2629            :                         cur_kv_len,
2630            :                         alibi_slopes,
2631            :                         score_output);
2632            :                 }
2633            : #    else
2634            :                 _helper.exec_kernel_multiple(
2635            :                     sub_query,
2636            :                     v_cache,
2637            :                     output_emb.slice(0, batch_in_token, batch_in_token + q_len)
2638            :                         .reshape({q_len, _helper.H * _helper.SV}),
2639            :                     _helper._qk_scratch_b.slice(0, batch_in_reorder, batch_in_reorder),
2640            :                     _helper._wv_scratch_b.slice(0, batch_in_reorder, batch_in_reorder),
2641            :                     block_indices.ptr<int32_t>() + block_indices_begins.ptr<int32_t>()[batch_in_seq],
2642            :                     ithr,
2643            :                     q_blk,
2644            :                     hq_beg,
2645            :                     hq_end,
2646            :                     hk,
2647            :                     q_len,
2648            :                     cur_kv_len,
2649            :                     alibi_slopes,
2650            :                     score_output);
2651            : #    endif
2652         4006 :             }
2653            :         });
2654           32 :         if (output_score) {
2655            :             parallel_for2d_dynamic(past_lens.m_dims[0], 1, [&](size_t b, size_t pq) {
2656            :                 auto seq_len = static_cast<size_t>(subsequence_begins.ptr<int32_t>()[b + 1] -
2657            :                                                    subsequence_begins.ptr<int32_t>()[b]);
2658            :                 auto cur_kv_len = static_cast<size_t>(past_lens.ptr<int32_t>()[b]) + seq_len;
2659            :                 auto src_offset = _helper._score_offsets_aligned.template ptr<int32_t>()[b];
2660            :                 auto* src = _helper._score_output.template ptr<float>() + src_offset * _helper.H;
2661            :                 size_t src_stride = rnd_up(cur_kv_len, 16);
2662            :                 auto dst_offset = _helper._score_offsets.template ptr<int32_t>()[b];
2663            :                 auto* dst = output_score.ptr<float>() + dst_offset;
2664            :                 attn_reduce(dst, src, _helper.H, cur_kv_len, src_stride);
2665            :             });
2666            :         }
2667           32 :     }
2668            :
2669            :     // Q, K, V is ready, do attention
2670         1280 :     void operator()(PlainTensor& query,
2671            :                     PlainTensor& present_key,
2672            :                     PlainTensor& present_value,
2673            :                     PlainTensor& output_emb,
2674            :                     PlainTensor& output_score,
2675            :                     size_t max_context_len,
2676            :                     const PlainTensor& past_lens,
2677            :                     const PlainTensor& subsequence_begins,
2678            :                     const PlainTensor& block_indices,
2679            :                     const PlainTensor& block_indices_begins,
2680            :                     const PlainTensor& alibi_slopes) {
2681         1280 :         _workitems.reset(query, past_lens, subsequence_begins, _helper._block_size);
2682         1280 :         if (output_score) {
2683            :             _helper.init_score_buffers(past_lens, subsequence_begins);
2684            :         }
2685            :
2686         1280 :         auto nthr = static_cast<size_t>(parallel_get_max_threads());
2687            :
2688         1280 :         if (past_lens.m_dims[0] >= nthr || _workitems.get_reorder_max_batch_size() > 0) {
2689           32 :             exec_loop_mixed(query,
2690            :                             present_key,
2691            :                             present_value,
2692            :                             output_emb,
2693            :                             output_score,
```

```
                                max_context_len,
                                past_lens,
                                subsequence_begins,
                                block_indices,
                                block_indices_begins,
                                alibi_slopes);
            } else {
1248            _helper.exec_loop_bhl(query,
                                      present_key,
                                      present_value,
                                      output_emb,
                                      output_score,
                                      max_context_len,
                                      past_lens,
                                      subsequence_begins,
                                      block_indices,
                                      block_indices_begins,
                                      alibi_slopes);
            }
1280    }
};

template <typename DATA_TYPE, typename KEY_CACHE_TYPE, ov::element::Type_t VALUE_PREC>
struct AttentionExecutor : public PagedAttentionExecutor {
    MHAHelper<DATA_TYPE, KEY_CACHE_TYPE, VALUE_PREC> _helper;
    MHA<DATA_TYPE, KEY_CACHE_TYPE, VALUE_PREC> _kernel;
    PlainTensor _slot_mapping;

    AttentionExecutor() : _kernel(_helper) {}

32  explicit AttentionExecutor(size_t key_group_size, size_t value_group_size, bool quant_key_bychannel)
32      : _helper(
                MHAHelper<DATA_TYPE, KEY_CACHE_TYPE, VALUE_PREC>(key_group_size, value_group_size, quant_key_bychannel)),
32        _kernel(_helper) {}

1280    void init(const std::vector<MemoryPtr>& inputs,
                  const std::vector<MemoryPtr>& outputs,
                  PlainTensor& q,
                  PlainTensor& k,
                  PlainTensor& v,
                  PlainTensor& k_cache,
                  PlainTensor& v_cache,
                  PlainTensor& past_lens,
                  PlainTensor& subsequence_begins,
                  PlainTensor& block_indices,
                  PlainTensor& block_indices_begins,
                  float& scale,
                  size_t& sliding_window,
                  PlainTensor& alibi_slopes,
                  size_t& max_context_len,
                  PlainTensor& rotated_block_indices,
                  PlainTensor& rotation_deltas,
                  PlainTensor& rotation_trig_lut,
                  PlainTensor& output_emb,
                  PlainTensor& output_score) {
1280        q.reset(inputs[ID_Q]);  // [B_token, H * S]
1280        k.reset(inputs[ID_K]);
1280        v.reset(inputs[ID_V]);
1280        k_cache.reset(inputs[ID_KCACHE]);                               // [NUM_BLOCKS, H, 32, S]
1280        v_cache.reset(inputs[ID_VCACHE]);                               // [NUM_BLOCKS, H, 32, S]
1280        past_lens.reset(inputs[ID_PAST_LENS]);                          // [B_seq]
1280        subsequence_begins.reset(inputs[ID_SUBSEQUENCE_BEGINS]);        // [B_seq+1]
1280        block_indices.reset(inputs[ID_BLOCK_INDICES]);                  // [num_blocks]
1280        block_indices_begins.reset(inputs[ID_BLOCK_INDICES_BEGINS]);    // [B_seq+1]
1280        scale = *inputs[ID_SCALE]->getDataAs<float>();
1280        sliding_window = static_cast<size_t>(*inputs[ID_SLIDING_WINDOW]->getDataAs<int32_t>());
1280        if (!inputs[ID_ALIBI_SLOPES]->getShape().hasZeroDims()) {
   0            alibi_slopes.reset(inputs[ID_ALIBI_SLOPES]);
            }
1280        max_context_len = static_cast<size_t>(*inputs[ID_MAX_CONTEXT_LEN]->getDataAs<int32_t>());

1280        size_t inputs_size = inputs.size();
1280        if (inputs_size > ID_ROTATED_BLOCK_INDICES) {
   0            OPENVINO_ASSERT(inputs_size >= ID_ROTATION_TRIG_LUT);
   0            if (!inputs[ID_ROTATED_BLOCK_INDICES]->getShape().hasZeroDims()) {
   0                rotated_block_indices.reset(inputs[ID_ROTATED_BLOCK_INDICES]);  // [num_blocks]
                }
   0            if (!inputs[ID_ROTATION_DELTAS]->getShape().hasZeroDims()) {
   0                rotation_deltas.reset(inputs[ID_ROTATION_DELTAS]);  // [num_blocks,  block_size (32) || 1]
                }
   0            if (!inputs[ID_ROTATION_TRIG_LUT]->getShape().hasZeroDims()) {
   0                rotation_trig_lut.reset(
                        inputs[ID_ROTATION_TRIG_LUT]);  // [max_context_len * embedding_size], row-major layout
                }
            }

1280        output_emb.reset(outputs[0]);
1280        if (outputs.size() == 2) {
   0            output_score.reset(outputs[1]);
            }

1280        auto B_token = q.size(0);
1280        auto Hk = k_cache.size(1);
            /* The layout for kv cache:

               by-hidden, quantized by S(hidden dims) group_num = N
               N * f32(scale + zp)|group_0|group_1|...|group_N
               adjusted_S = S + N * f32(scale + zp) * sub_byte_multiplier

               by channel, quantized by channel [block_size, S], group_num = block_size
               adjusted block consists 3 parts
               f32[scale, S]
               f32[zp, S]
               u8[block_size, S]
```

```
2798             :             adjusted key cache block u8[block_size + 2 * sizeof(float) * sub_byte_multiplier, S]
2799             :         */
2800             :
2801        1280 :         const size_t key_sub_byte_multiplier = get_sub_byte_multiplier(k_cache.get_precision());
2802        1280 :         const size_t value_sub_byte_multiplier = get_sub_byte_multiplier(v_cache.get_precision());
2803        1280 :         const size_t key_params_size = sizeof(float) * 2 * key_sub_byte_multiplier;
2804             :         // u4 needs scale + zp. s4 needs scale.
2805        1280 :         const size_t param_size =
2806        2560 :             one_of(v_cache.get_precision(), ov::element::u4, ov::element::u8) ? sizeof(float) * 2 : sizeof(float);
2807        1280 :         const size_t value_params_size = param_size * value_sub_byte_multiplier;
2808        1280 :         size_t key_group_num =
2809        1280 :             _helper._key_group_size ? k_cache.size(3) / (_helper._key_group_size + key_params_size) : 1;
2810        1280 :         size_t value_group_num =
2811        1280 :             _helper._value_group_size ? v_cache.size(3) / (_helper._value_group_size + value_params_size) : 1;
2812        1280 :         auto SV = v_cache.size(3) - (v_cache.get_precision().is_real() ? 0 : value_params_size * value_group_num);
2813             :         // revise group_size if it's zero.
2814        1280 :         _helper._value_group_size = _helper._value_group_size ? _helper._value_group_size : SV;
2815             :
2816             :         // check by_hidden_dims parameter of value cache
2817        1280 :         if (!value_group_num && v_cache.get_precision().is_integral())
2818           0 :             OPENVINO_THROW("PagedAttn value cache gets wrong group_size, ",
2819             :                            _helper._value_group_size,
2820             :                            " should be smaller than hidden_dims");
2821        1280 :         size_t S = 0;
2822        1280 :         if (_helper._quant_key_bychannel) {
2823           0 :             S = k_cache.size(3);
2824             :         } else {
2825             :             // check by_hidden_dims parameter of key cache
2826        1280 :             if (!key_group_num && k_cache.get_precision().is_integral())
2827           0 :                 OPENVINO_THROW("PagedAttn key cache gets wrong group_size, ",
2828             :                                _helper._key_group_size,
2829             :                                " should be smaller than hidden_dims");
2830        1280 :             S = k_cache.size(3) - (k_cache.get_precision().is_real() ? 0 : key_params_size * key_group_num);
2831        2528 :             _helper._key_group_size = _helper._key_group_size ? _helper._key_group_size : S;
2832             :         }
2833        1280 :         auto block_size = _helper._quant_key_bychannel ? (k_cache.size(2) - key_params_size) : k_cache.size(2);
2834        1280 :         auto H = q.size(1) / S;
2835        1280 :         auto h_each_group_len = 1;
2836        1280 :         if (Hk != H) {
2837           0 :             h_each_group_len = H / Hk;
2838             :         }
2839        1280 :         auto B_seq = past_lens.size(0);
2840             :
2841        1280 :         q.assert_dims({B_token, H * S});
2842        1280 :         k.assert_dims({B_token, Hk * S});
2843        1280 :         v.assert_dims({B_token, Hk * SV});
2844        1280 :         q = q.reshape({B_token, H, 1, S});
2845        1280 :         k = k.reshape({B_token, Hk, 1, S});
2846        1280 :         v = v.reshape({B_token, Hk, 1, SV});
2847             :
2848        1280 :         if (k_cache.m_dt == ov::element::Type_t::u8) {
2849        1280 :             if (_helper._quant_key_bychannel) {
2850           0 :                 k_cache.assert_dims({0, Hk, block_size + key_params_size, S}, true);
2851             :             } else {
2852        1280 :                 k_cache.assert_dims({0, Hk, block_size, S + key_params_size * key_group_num}, true);
2853             :             }
2854        1280 :             v_cache.assert_dims({k_cache.m_dims[0], Hk, block_size, SV + value_params_size * value_group_num});
2855             :
2856             :         } else {
2857           0 :             k_cache.assert_dims({0, Hk, block_size, S}, true);
2858           0 :             v_cache.assert_dims({k_cache.m_dims[0], Hk, block_size, SV});
2859             :         }
2860        1280 :         past_lens.assert_dims({B_seq});
2861        1280 :         subsequence_begins.assert_dims({B_seq + 1});
2862        1280 :         block_indices.assert_dims({0}, true);
2863        1280 :         block_indices_begins.assert_dims({B_seq + 1});
2864        1280 :         if (scale == 0.0f) {
2865           0 :             scale = 1.0f / sqrt(S);
2866             :         }
2867        1280 :         if (alibi_slopes) {
2868           0 :             alibi_slopes.assert_dims({H});
2869             :         }
2870             :
2871        1280 :         bool init_rotation_coefficient_scratch = false;
2872        1280 :         if (rotated_block_indices) {
2873             :             // Only K entries are needed to be rotated, since position is encoded at the Q^T @ (effective_RoPE_matrix) @
2874             :             // K matrix multiplication
2875           0 :             rotation_deltas.assert_dims({rotated_block_indices.size(0), 0}, /* special_zero = */ true);
2876           0 :             OPENVINO_ASSERT(rotation_deltas.shape()[1] == 1 ||
2877             :                             rotation_deltas.shape()[1] == block_size);  // per-block or per-token granularity
2878           0 :             rotation_trig_lut.assert_dims({0, S}, /* special_zero = */ true);
2879           0 :             init_rotation_coefficient_scratch = true;
2880             :         }
2881        1280 :         output_emb.assert_dims({B_token, H * SV});
2882        1280 :         output_emb = output_emb.reshape({B_token, 1, H * SV});
2883             :
2884             :         // TODO: enable block_size to be multiple of 32
2885        1280 :         OPENVINO_ASSERT(block_size == 32, "CPU: block size must be 32, current: ", block_size);
2886             :
2887        1280 :         _helper.init(H,
2888             :                      S,
2889             :                      SV,
2890             :                      Hk,
2891             :                      h_each_group_len,
2892             :                      block_size,
2893             :                      sliding_window,
2894             :                      scale,
2895             :                      max_context_len,
2896             :                      alibi_slopes,
2897             :                      init_rotation_coefficient_scratch);
2898        1280 :     }
2899             :
2900        1280 :     void concat_pastkv(const PlainTensor& k,
2901             :                        const PlainTensor& v,
```

```
                                    PlainTensor& k_cache,
                                    PlainTensor& v_cache,
                                    const PlainTensor& past_lens,
                                    const PlainTensor& subsequence_begins,
                                    const PlainTensor& block_indices,
                                    const PlainTensor& block_indices_begins) {
        1280          auto B_token = k.size(0);
        2560          _slot_mapping.resize<int32_t>({B_token});

        1280          size_t idx = 0;
        6400          for (size_t i = 0; i < past_lens.size(0); i++) {
        5120              auto q_len = subsequence_begins.ptr<int32_t>()[i + 1] - subsequence_begins.ptr<int32_t>()[i];
        5120              auto kv_len = past_lens.ptr<int32_t>()[i] + q_len;
        5120              auto block_number_start = block_indices_begins.ptr<int32_t>()[i];
        5120              auto block_offset_start = kv_len - q_len;
       13280              for (int32_t j = 0; j < q_len; j++) {
        8160                  auto block_offset = block_offset_start + j;
        8160                  auto block_number =
        8160                      block_indices.ptr<int32_t>()[block_number_start + block_offset / _helper._block_size];
        8160                  _slot_mapping.ptr<int32_t>()[idx++] =
        8160                      block_number * _helper._block_size + block_offset % _helper._block_size;
                          }
                          // To simplify tails of the kernels for Q*K and W*V:
                          // for first token kernels:
                          //    Q*K aka [m, k0] * [n0, k0]', there is already tails logic for m, k0, but no(always assume n0 is
                          //    block_size) for n0 which means the tail of k_cache need to be set to zero.
                          //    W*V aka [m, k1] * [n1, k1]', there is no tails handing for n1, so tails of v_cache need to be set to
                          //    zero.
                          // for second token, the kernels have tails handling logic
        5120              if (q_len != 1 && kv_len % _helper._block_size != 0) {
                              // block no. which contains tails
         128                  auto block_number = block_indices.ptr<int32_t>()[block_number_start + kv_len / _helper._block_size];
                              // tails start position
         128                  auto block_offset = kv_len % _helper._block_size;
                              // tails number
         128                  auto zero_tokens = _helper._block_size - block_offset;
         128                  auto S = k_cache.m_dims[3];
         128                  auto SV = v_cache.m_dims[3];
         128                  auto Hk = k_cache.m_dims[1];  // shape: [block, H, 32, S]
       17256                  parallel_for2d(Hk, zero_tokens, [&](size_t h, size_t l) {
                                  auto set_zero =
        7444                          [](PlainTensor& cache, size_t block_number, size_t h, size_t l, size_t hidden_dims) {
        7444                              auto sub_byte_multiplier = get_sub_byte_multiplier(cache.get_precision());
        8034                              size_t cache_stride =
        7245                                  (block_number * cache.stride(0) + h * cache.stride(1) + l * cache.stride(2)) *
        7191                                  cache.get_precision().size() / sub_byte_multiplier;
        8034                              auto cache_ptr = cache.m_ptr.get() + cache_stride;
        8034                              std::memset(cache_ptr, 0, hidden_dims * cache.m_element_size / sub_byte_multiplier);
                                      };
        5394                          set_zero(k_cache, block_number, h, block_offset + l, S);
        5776                          set_zero(v_cache, block_number, h, block_offset + l, SV);
                                  });
                              }
                          }

        1280          if (k_cache.m_dt == ov::element::Type_t::u8) {
                          // slot_mapping could only be used for per token quantization
                          // by_channel needs all data to calculation block info.
        1280              paged_attn_quantkv(k,
                                             v,
                                             k_cache,
                                             v_cache,
                                             past_lens,
                                             subsequence_begins,
                                             block_indices,
                                             block_indices_begins,
                                             _slot_mapping,
        1280                                 _helper._output,
                                             _helper._quant_key_bychannel,
                                             _helper._key_group_size,
                                             _helper._value_group_size);
                          } else {
           0                  paged_attn_memcpy(k, v, k_cache, v_cache, _slot_mapping);
                          }
        1280      }

        1280      void execute(const std::vector<MemoryPtr>& inputs, const std::vector<MemoryPtr> outputs) override {
        1280          PlainTensor q, k, v, k_cache, v_cache;
        1280          PlainTensor past_lens, subsequence_begins, block_indices, block_indices_begins;
                      float scale;
                      size_t sliding_window;
        1280          PlainTensor alibi_slopes;
                      size_t max_context_len;
        1280          PlainTensor rotated_block_indices;
        1280          PlainTensor rotation_deltas;
        1280          PlainTensor rotation_trig_lut;

        1280          PlainTensor output_emb;
        1280          PlainTensor output_score;

        1280          init(inputs,
                           outputs,
                           q,
                           k,
                           v,
                           k_cache,
                           v_cache,
                           past_lens,
                           subsequence_begins,
                           block_indices,
                           block_indices_begins,
                           scale,
                           sliding_window,
                           alibi_slopes,
```

```
3006             :             max_context_len,
3007             :             rotated_block_indices,
3008             :             rotation_deltas,
3009             :             rotation_trig_lut,
3010             :             output_emb,
3011             :             output_score);
3012             :
3013        1280 :         if (rotated_block_indices) {
3014           0 :             rotate_kv_cache<KEY_CACHE_TYPE>(k_cache,
3015             :                                             rotated_block_indices,
3016             :                                             rotation_deltas,
3017             :                                             rotation_trig_lut,
3018           0 :                                             _helper._block_rotation_coefficient_scratch);
3019             :         }
3020             :
3021        1280 :         concat_pastkv(k, v, k_cache, v_cache, past_lens, subsequence_begins, block_indices, block_indices_begins);
3022             :
3023        1280 :         _kernel(q,
3024             :                 k_cache,
3025             :                 v_cache,
3026             :                 output_emb,
3027             :                 output_score,
3028             :                 max_context_len,
3029             :                 past_lens,
3030             :                 subsequence_begins,
3031             :                 block_indices,
3032             :                 block_indices_begins,
3033             :                 alibi_slopes);
3034        1280 :     }
3035             : };
3036             : #endif
3037             :
3038          32 : std::shared_ptr<PagedAttentionExecutor> make_pa_executor(ov::element::Type data_type,
3039             :                                                          ov::element::Type key_cache_type,
3040             :                                                          ov::element::Type value_cache_type,
3041             :                                                          size_t key_group_size,
3042             :                                                          size_t value_group_size,
3043             :                                                          bool quant_key_bychannel) {
3044          32 :     std::shared_ptr<PagedAttentionExecutor> executor;
3045             : #if defined(OPENVINO_ARCH_X86_64)
3046             :     if (data_type == ov::element::bf16) {
3047             : #    if defined(HAVE_AVX512F)
3048             :         if (key_cache_type == ov::element::u8) {
3049             :             if (value_cache_type == ov::element::u4) {
3050             :                 executor =
3051             :                     std::make_shared<AttentionExecutor<ov::bfloat16, uint8_t, ov::element::u4>>(key_group_size,
3052             :                                                                                                 value_group_size,
3053             :                                                                                                 quant_key_bychannel);
3054             :             } else if (value_cache_type == ov::element::u8) {
3055             :                 executor =
3056             :                     std::make_shared<AttentionExecutor<ov::bfloat16, uint8_t, ov::element::u8>>(key_group_size,
3057             :                                                                                                 value_group_size,
3058             :                                                                                                 quant_key_bychannel);
3059             :             } else {
3060             :                 OPENVINO_THROW("make_pa_executor: key_cache_type u8 with value_cache_type ",
3061             :                                value_cache_type.to_string(),
3062             :                                " is not support");
3063             :             }
3064             :
3065             :         } else {
3066             :             OPENVINO_ASSERT(key_cache_type == ov::element::bf16, "expect kvcache type bf16, current: ", key_cache_type);
3067             :             executor = std::make_shared<AttentionExecutor<ov::bfloat16, ov::bfloat16, ov::element::bf16>>();
3068             :         }
3069             : #    else
3070             :         OPENVINO_THROW("make_pa_executor: bf16 needs avx512+ hardware.");
3071             : #    endif
3072             :     } else if (data_type == ov::element::f16) {
3073             : #    if defined(HAVE_AVX512F)
3074             :         if (key_cache_type == ov::element::u8) {
3075             :             if (value_cache_type == ov::element::u4) {
3076             :                 executor =
3077             :                     std::make_shared<AttentionExecutor<ov::float16, uint8_t, ov::element::u4>>(key_group_size,
3078             :                                                                                                value_group_size,
3079             :                                                                                                quant_key_bychannel);
3080             :             } else if (value_cache_type == ov::element::u8) {
3081             :                 executor =
3082             :                     std::make_shared<AttentionExecutor<ov::float16, uint8_t, ov::element::u8>>(key_group_size,
3083             :                                                                                                value_group_size,
3084             :                                                                                                quant_key_bychannel);
3085             :             } else {
3086             :                 OPENVINO_THROW("make_pa_executor: key_cache_type u8 with value_cache_type ",
3087             :                                value_cache_type.to_string(),
3088             :                                " is not support");
3089             :             }
3090             :         } else {
3091             :             OPENVINO_ASSERT(key_cache_type == ov::element::f16, "expect kvcache type f16, current: ", key_cache_type);
3092             :             executor = std::make_shared<AttentionExecutor<ov::float16, ov::float16, ov::element::f16>>();
3093             :         }
3094             : #    else
3095             :         OPENVINO_THROW("make_pa_executor: f16 needs avx512+ hardware.");
3096             : #    endif
3097             :     } else if (data_type == ov::element::f32) {
3098             :         if (key_cache_type == ov::element::u8) {
3099             :             if (value_cache_type == ov::element::u4) {
3100             :                 executor = std::make_shared<AttentionExecutor<float, uint8_t, ov::element::u4>>(key_group_size,
3101             :                                                                                                 value_group_size,
3102             :                                                                                                 quant_key_bychannel);
3103             :             } else if (value_cache_type == ov::element::u8) {
3104             :                 executor = std::make_shared<AttentionExecutor<float, uint8_t, ov::element::u8>>(key_group_size,
3105             :                                                                                                 value_group_size,
3106             :                                                                                                 quant_key_bychannel);
3107             :             } else {
3108             :                 OPENVINO_THROW("make_pa_executor: key_cache_type u8 with value_cache_type ",
3109             :                                value_cache_type.to_string(),
```

```
3110             :                                 " is not support");
3111             :             }
3112             :         } else if (key_cache_type == ov::element::f16) {
3113             :             executor = std::make_shared<AttentionExecutor<float, ov::float16, ov::element::f16>>(key_group_size,
3114             :                                                                                                  value_group_size,
3115             :                                                                                                  quant_key_bychannel);
3116             :         } else {
3117             :             OPENVINO_ASSERT(key_cache_type == ov::element::f32, "expect kvcache type f32, current: ", key_cache_type);
3118             :             executor = std::make_shared<AttentionExecutor<float, float, ov::element::f32>>(key_group_size,
3119             :                                                                                            value_group_size,
3120             :                                                                                            quant_key_bychannel);
3121             :         }
3122             :     } else {
3123             :         OPENVINO_THROW("make_pa_executor: unsupported precision: ", data_type);
3124             :     }
3125             : #elif (defined(OPENVINO_ARCH_ARM64) && defined(HAVE_SVE))
3126          32 :     if (data_type == ov::element::f32) {
3127           0 :         if (key_cache_type == ov::element::u8 && value_cache_type == ov::element::u8) {
3128           0 :             executor = std::make_shared<AttentionExecutor<float, uint8_t, ov::element::u8>>(key_group_size,
3129             :                                                                                             value_group_size,
3130           0 :                                                                                             quant_key_bychannel);
3131             :         } else {
3132           0 :             OPENVINO_THROW("make_pa_executor: key_cache_type and value_cache_type of u8 is only support");
3133             :         }
3134             :     }
3135          32 :     if (data_type == ov::element::f16) {
3136          32 :         if (key_cache_type == ov::element::u8 && value_cache_type == ov::element::u8) {
3137          32 :             executor = std::make_shared<AttentionExecutor<ov::float16, uint8_t, ov::element::u8>>(key_group_size,
3138             :                                                                                                  value_group_size,
3139          32 :                                                                                                  quant_key_bychannel);
3140             :         } else {
3141          32 :             OPENVINO_THROW("make_pa_executor: key_cache_type and value_cache_type of u8 is only support");
3142             :         }
3143             :     }
3144             :
3145             : #else
3146             :     OPENVINO_THROW("make_pa_executor: only support x64 platform or ARM with SVE support");
3147             : #endif
3148          32 :     return executor;
3149           0 : }
3150             :
3151             : }  // namespace ov::Extensions::Cpu::XARCH
```

Generated by: LCOV version 2.3-beta