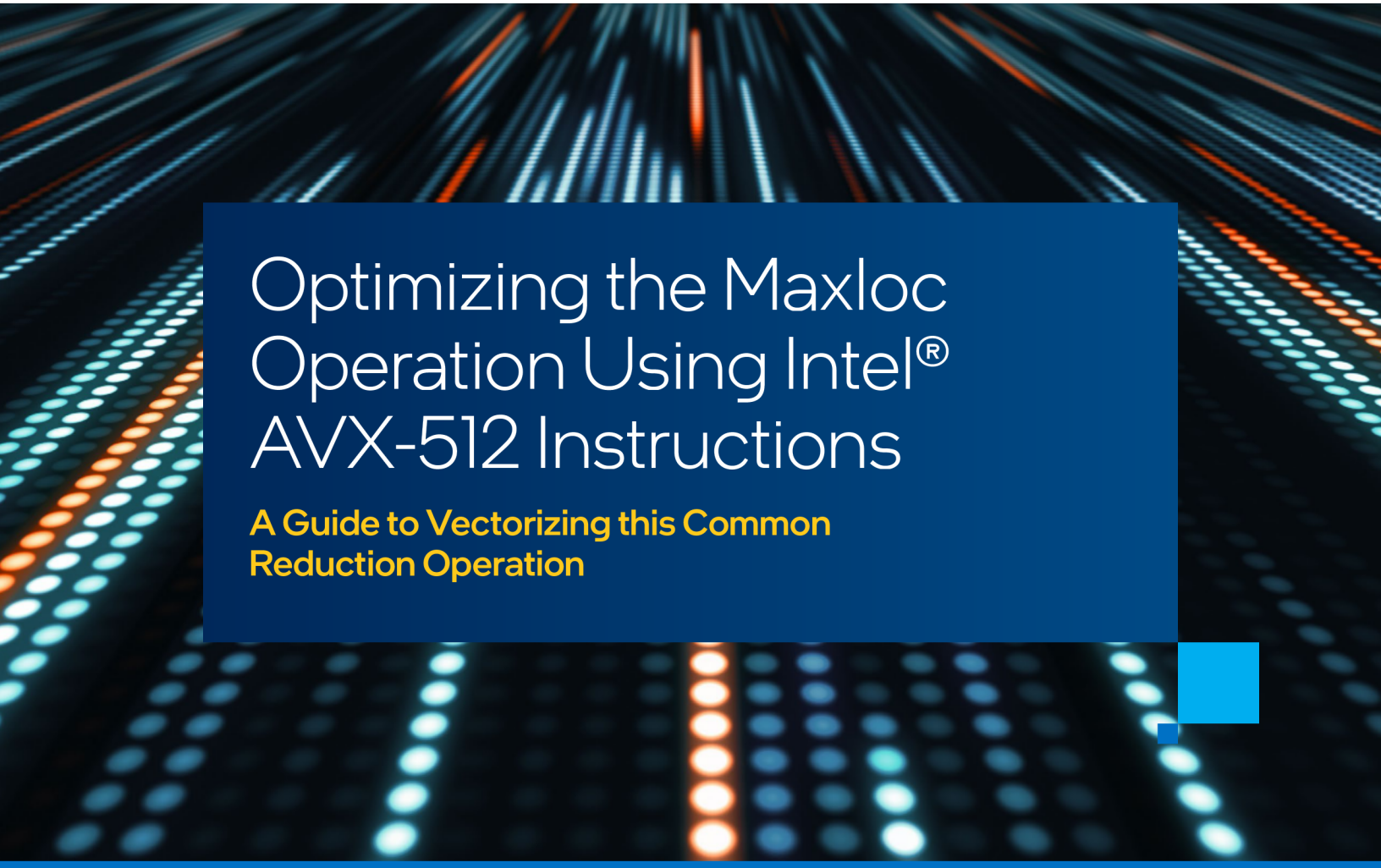

Optimizing the Maxloc Operation Using Intel® AVX-512 Instructions

A Guide to Vectorizing this Common Reduction Operation

Vamsi Sripathi, Senior HPC Application Engineer, Intel Corporation

Finding the index of a target element (minimum, maximum, or their absolute values) is used in many applications. In this article, I use the Intel® Advanced Vector Extensions 512 (Intel AVX-512) compiler intrinsics to accelerate this operation (henceforth referred to as "Maxloc") and benchmark the performance on Intel® Xeon® Scalable processors. Intel AVX-512 provides a broad set of instructions that facilitates single instruction, multiple data (SIMD) execution. It is available on all Intel Xeon Scalable processors and uses 512-bit vector registers to operate on wider execution units for maximum efficiency. By careful application of Intel AVX-512, the number of instructions and comparisons needed to identify the index of the target element is minimized, resulting in a significant speed-up.

Maxloc is a common search operation performed on arrays (e.g., the NumPy argmax function, TensorFlow

GlobalMaxPool1D, and oneMKL amax functions). I will be using the Intel AVX-512 SIMD API (popularly known as vector *intrinsics*) provided by the Intel® C/C++ Compiler Classic. (Readers are also referred to my previous article, Optimization of Scan Operations Using Explicit Vectorization.) The following sections will illustrate the implementation of Intel AVX-512 Maxloc using Intel AVX-512 vector intrinsics.

Baseline Implementation

Code listing 1 shows a baseline implementation consisting of a single pass, scalar version of Maxloc. As can be seen from the code, a vector containing n elements requires n comparison operations. We visit the input vector only once, and store the index location corresponding to the maximum value. The instructions generated in the baseline implementation are shown in **Table 2** and will be discussed in the performance evaluation section.

```
1 #include "float.h"
2
3 int ref_max (int *p_n, float *p_x)
4 {
5   int n = *p_n;
6   float max_val = FLT_MIN;
7   int idx;
8
9   for (int i=0; i<n; i++) {
10    if (p_x[i] > max_val) {
11      max_val = p_x[i];
12      idx = i;
13    }
14  }
15  return idx;
16 }
```

Code listing 1: Maxloc baseline implementation

Intel AVX-512 SIMD Implementation

Table 1 lists the Intel AVX-512 SIMD instructions used in my approach. **Figure 1** shows a visual representation of the operations performed by `vmaxpd`, `vcmpps`, and `vblendmps` instructions for a sample register state.

Op Name	Instruction	AVX512 Vector Intrinsics API	Description
SIMD Load	vmovups	__m512d _mm512_loadu_ps (void *mem_address);	Load 512-bits (composed of 16 packed single-precision (32-bit) floating-point elements) from memory into dst.
SIMD Max	vmaxps	__m512d _mm512_max_ps (__m512d a, __m512d b);	Compare packed single-precision (32-bit) floating-point elements in a and b, and store packed maximum values in dst.
SIMD Compare	vcmpps	__mmask16 _mm512_cmp_ps_mask (__m512 a, __m512 b, const int imm8);	Compare packed single-precision (32-bit) floating-point elements in a and b based on the comparison operand specified by imm8, and store the results in mask vector k.
SIMD Blend	vblendmps	__m512 _mm512_mask_blend_ps (__mmask16 k, __m512 a, __m512 b);	Blend packed single-precision (32-bit) floating-point elements from a and b using control mask k, and store the results in dst.
SIMD Broadcast	vpbroadcastd	__m512i _mm512_mask_set1_epi32 (__m512i src, __mmask16 k, int a);	Broadcast 32-bit integer a to all elements of dst using writemask k (elements are copied from src when the corresponding mask bit is not set).
SIMD Extract	vextractf32x8	__m256 _mm512_extractf32x8_ps (__m512 a, int imm8);	Extract 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from a, selected with imm8, and store the result in dst.

Table 1. Intel AVX-512 SIMD instructions used in Maxloc

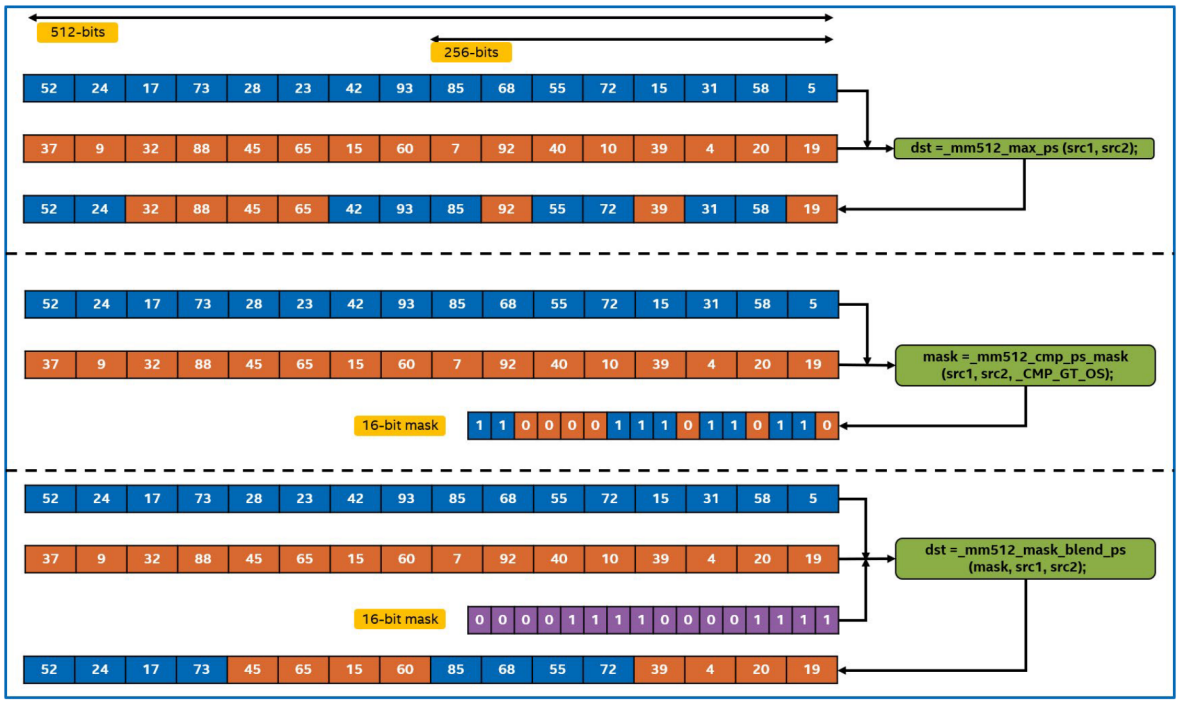

Figure 1. Visual representation of operations performed by Intel AVX-512 `vmaxps, vcmpps, and vblendmps`

For Maxloc, I unroll the loop and apply a series of `vmaxpd` instructions to obtain a sequence of 16 FP32 (32-bit floating-point) maximum values, represented as 512-bit Intel AVX-512 registers. In addition to that, I keep track of the indices of the maximum values as the input data is processed. This is achieved by doing an additional comparison operation (between the current and previous iteration maximum values) and using the results to blend the latest maximum values and their corresponding unrolled block ID. The key steps of the algorithm are as follows:

1. **Main block:**
    a. Each iteration loads 64 input elements into four Intel AVX-512 vector registers and does pairwise comparison to identify the maximum value in each quadruplet to form a current max register representing the max values in offsets of 16.
    b. The previously populated max values are compared to form a 16-bit mask (with each bit representing a FP32 element in an Intel AVX-512 register). The mask bit is set if the quadruplet maximum value found in the current iteration is greater than the previous quadruplet max value.
    c. Once this mask is calculated, it is used to blend the values from both the current and previous max Intel AVX-512 registers to form the new quadruplet of max values in the register for use in future iterations.
    d. To track the indices, the same mask is reused to set the current loop index (unrolled block ID) to a register of indices representing the block at which the corresponding maximum value is located.
    e. The previous steps form the main loop and are repeated for all input elements to find the 16 max values among quadruplets (at indices {0,16,32,48}, {1,17,33,49}, ...) and their corresponding loop/block IDs (0, 64, 128, 192, ...).

2. **Reduction and target block:** After the main loop, a single reduction on an Intel AVX-512 register finds the maximum among the 16 values from the main block. The maximum value found from the reduction step is used to identify (in the form of a 16-bit mask) where among the 16 locations in the Intel AVX-512 register the maximum value is encountered. Each bit of the 16-bit result mask represents a selection of int32 elements into the Intel AVX-512 block-indices register. This int32 value represents the iteration ID where the maximum value is encountered. Now, depending on whether the maximum value is encountered at one or more locations, the following two methods identify the first block containing the maximum value.
3. **Single instance:** In this case, the maximum value is found only once in the dataset, which implies a single set bit in the 16b result mask. This identifies the corresponding int32 value from the Intel AVX-512 block-indices register. For example, when the mask is 0000 0000 1000 0000, the value at the seventh int32 element in the Intel AVX-512 indices register represents the block ID where the maximum value is found.

   Offset into block: One of the unique features of this implementation is that we can use the inherent pattern of the `vmaxps` instruction to identify the target index without reading the whole block (size of N_UNROLL). Let's denote Blk_{id} as the int32 value from the Intel AVX-512 indices register corresponding to the 1-bit set in the 16-bit result mask, M_x as the bit location (0 to 15) where 1-bit is set in in 16-bit result mask. From **Figure 2**, we observe that `vmaxps` compares values that are at offsets of 16 elements. Hence, we can conclude that index of maximum value has occurred at one of the following four locations: ($Blk_{id} + M_x$), ($Blk_{id} + M_x + 16$), ($Blk_{id} + M_x + 32$), ($Blk_{id} + M_x + 48$) when N_UNROLL is 64.
4. **Duplicates:** If the number of 1-bits in the result mask is greater than one, then the maximum value is encountered at more than one location in the dataset. So, we find the minimum of all the int32 values corresponding to the 1-bits set in the mask from the Intel AVX-512 indices register. We again compare the minimum index against the Intel AVX-512 indices register to know if the maximum value is encountered more than once within an unrolled block. If true, we read the whole block in chunks of 16 elements to identify the first index where the maximum value is encountered. Otherwise, we use the approach described in Step 3.

Figures 2 and **3** show the visual representation of main block computations and reduction sequence on Intel AVX-512 registers. **Code listing 2** shows the implementation of this algorithm.

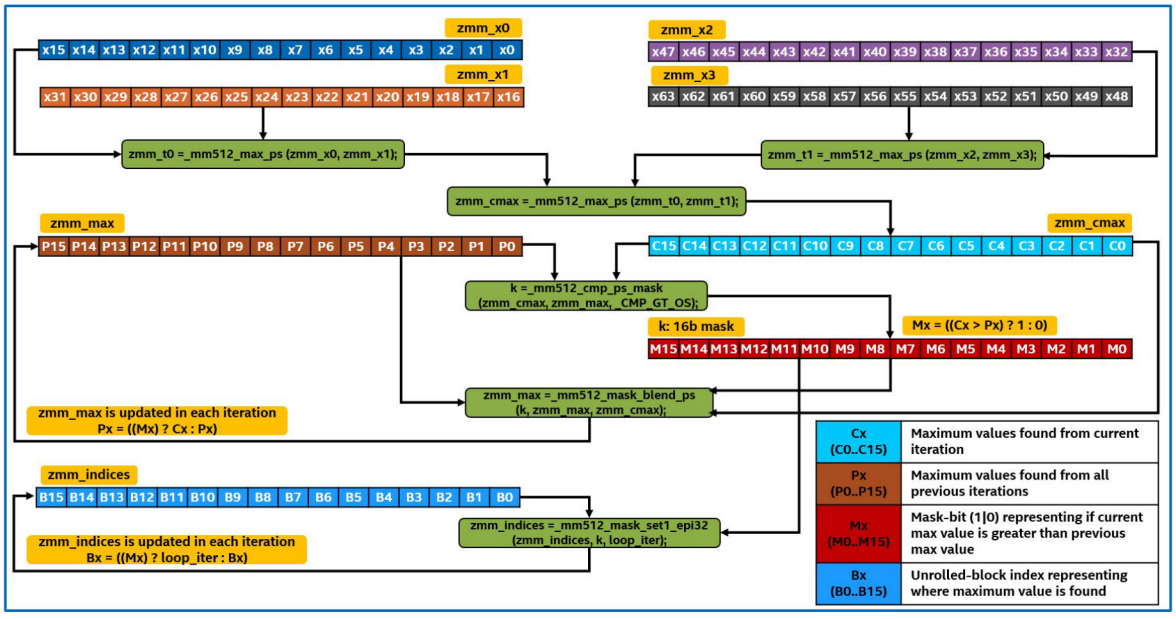

Figure 2. Visual representation of the main block of AVX-512 + Index Tracking implementation of Maxloc

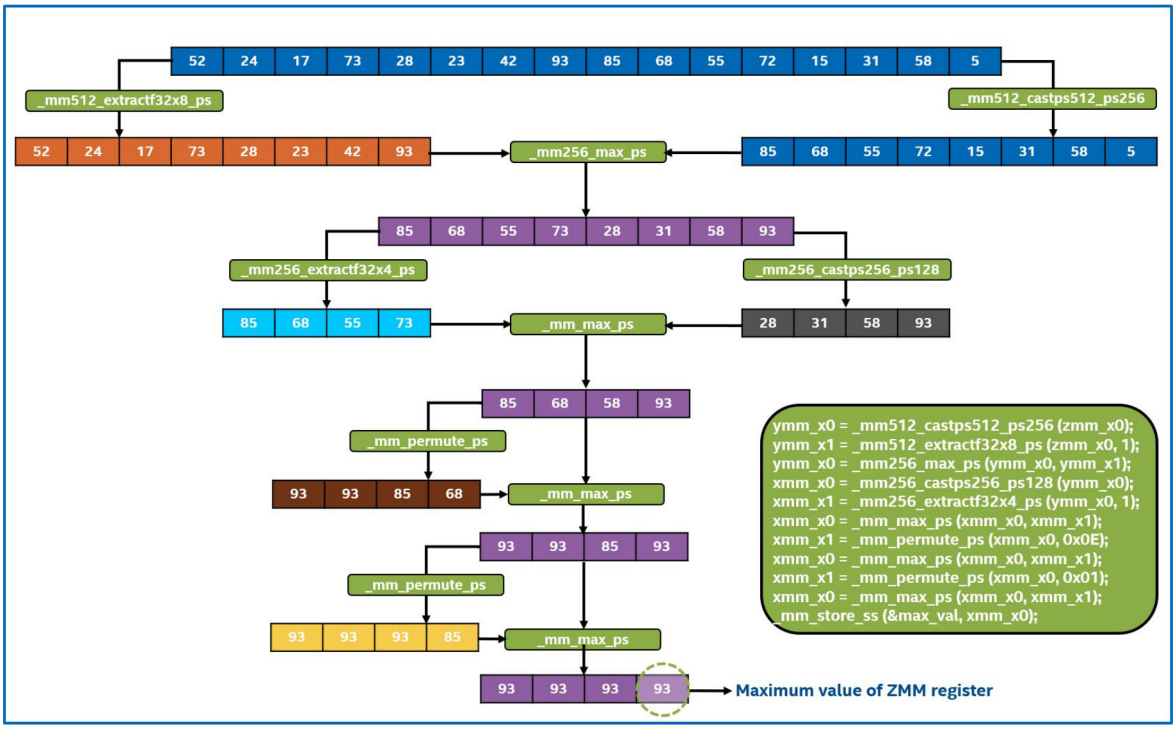

Figure 3. Instruction sequence to reduce Intel AVX-512/ZMM register to find the maximum value

```
int max_idx_tracking (int *p_n, float *p_x)
{
  __m512 zmm_x0, zmm_x1, zmm_x2, zmm_x3, zmm_x4, zmm_x5, zmm_x6, zmm_x7;
  __m512 zmm_curr_max, zmm_tmp0, zmm_tmp1, zmm_tmp2, zmm_tmp3, zmm_tmp4, zmm_tmp5;
  __m256 ymm_x0, ymm_x1;
  __m128 xmm_x0, xmm_x1;
  __mmask16 m0;
  __m512 zmm_max      = _mm512_set1_ps(FLT_MIN);
  __m512i zmm_indices = _mm512_setzero_epi32();

  int n = *p_n;
  int offset, block_idx;
  float max_val;

  for (int j=0; j<n; j+=N_UNROLL) {
    zmm_x0   = _mm512_loadu_ps(p_x);
    zmm_x1   = _mm512_loadu_ps(p_x+16);
    zmm_x2   = _mm512_loadu_ps(p_x+32);
    zmm_x3   = _mm512_loadu_ps(p_x+48);
    zmm_tmp0 = _mm512_max_ps(zmm_x0, zmm_x1);
    zmm_tmp1 = _mm512_max_ps(zmm_x2, zmm_x3);
#if N_UNROLL > 64
    zmm_x4   = _mm512_loadu_ps(p_x+64);
    zmm_x5   = _mm512_loadu_ps(p_x+80);
    zmm_x6   = _mm512_loadu_ps(p_x+96);
    zmm_x7   = _mm512_loadu_ps(p_x+112);
    zmm_tmp2 = _mm512_max_ps(zmm_x4, zmm_x5);
    zmm_tmp3 = _mm512_max_ps(zmm_x6, zmm_x7);
    zmm_tmp4 = _mm512_max_ps(zmm_tmp0, zmm_tmp1);
    zmm_tmp5 = _mm512_max_ps(zmm_tmp2, zmm_tmp3);

    zmm_curr_max = _mm512_max_ps(zmm_tmp4, zmm_tmp5);
#else
    zmm_curr_max = _mm512_max_ps(zmm_tmp0, zmm_tmp1);
#endif

    m0 = _mm512_cmp_ps_mask(zmm_curr_max, zmm_max, _CMP_GT_OS);
    if (m0) {
      zmm_max      = _mm512_mask_blend_ps(m0, zmm_max, zmm_curr_max);
      zmm_indices  = _mm512_mask_set1_epi32(zmm_indices, m0, j);
    }
    p_x += N_UNROLL;
  }

  p_x = p_x-n;

  // reduction tree on zmm register to find max. value
  ymm_x0 = _mm512_castps512_ps256(zmm_max);
  ymm_x1 = _mm512_extractf32x8_ps(zmm_max, 1);
  ymm_x0 = _mm256_max_ps(ymm_x0, ymm_x1);

  xmm_x0 = _mm256_castps256_ps128(ymm_x0);
  xmm_x1 = _mm256_extractf32x4_ps(ymm_x0, 1);
  xmm_x0 = _mm_max_ps(xmm_x0, xmm_x1);

  xmm_x1 = _mm_permute_ps(xmm_x0, 0x0E);
  xmm_x0 = _mm_max_ps(xmm_x0, xmm_x1);

  xmm_x1 = _mm_permute_ps(xmm_x0, 0x01);
  xmm_x0 = _mm_max_ps(xmm_x0, xmm_x1);
  _mm_store_ss(&max_val, xmm_x0);

  m0 = _mm512_cmp_ps_mask(_mm512_broadcastss_ps(xmm_x0), zmm_max, _CMP_EQ_OQ);
  int pop_cnt = _mm_popcnt_u32(_cvtmask16_u32(m0));
  if (pop_cnt == 1) {
    _mm512_mask_compressstoreu_epi32(&block_idx, m0, zmm_indices);
    offset = _bit_scan_forward(m0);
  } else {
    __m256i ymm_i0, ymm_i1;
    __m128i xmm_i0, xmm_i1;

    zmm_indices = _mm512_mask_blend_epi32(m0, _mm512_set1_epi32(INT_MAX), zmm_indices);
    // reduction tree on zmm register to find min. index value
    ymm_i0 = _mm512_castsi512_si256(zmm_indices);
    ymm_i1 = _mm512_extracti32x8_epi32(zmm_indices, 1);
    ymm_i0 = _mm256_min_epi32(ymm_i0, ymm_i1);

    xmm_i0 = _mm256_castsi256_si128(ymm_i0);
    xmm_i1 = _mm256_extracti32x4_epi32(ymm_i0, 1);
    xmm_i0 = _mm_min_epi32(xmm_i0, xmm_i1);

    xmm_i1 = _mm_shuffle_epi32(xmm_i0, 0x0E);
    xmm_i0 = _mm_min_epi32(xmm_i0, xmm_i1);

    xmm_i1 = _mm_shuffle_epi32(xmm_i0, 0x01);
    xmm_i0 = _mm_min_epi32(xmm_i0, xmm_i1);
    _mm_store_ss((float *)&block_idx, _mm_castsi128_ps(xmm_i0));

    __mmask16 m1 = _mm512_cmp_epi32_mask(
                   _mm512_broadcastd_epi32(xmm_i0), zmm_indices, _MM_CMPINT_EQ);
    if (_mm_popcnt_u32(_cvtmask16_u32(m1)) > 1) {
      p_x += block_idx;
      zmm_max = _mm512_broadcastss_ps(xmm_x0);

      for (int j=0; j<N_UNROLL; j+=NUM_ELEMS_IN_REG) {
        zmm_x0 = _mm512_loadu_ps(p_x);
        m0     = _mm512_cmp_ps_mask(zmm_x0, zmm_max, _CMP_EQ_OQ);
        if (m0) {
          return (block_idx + j + _bit_scan_forward(m0));
        }
        p_x += NUM_ELEMS_IN_REG;
      }
    }
    offset = _bit_scan_forward(m1);
  }

  for (int i=0; i<(N_UNROLL/NUM_ELEMS_IN_REG); i++) {
    if (max_val == p_x[block_idx+offset+(i*NUM_ELEMS_IN_REG)]) {
      return block_idx + offset + (i*NUM_ELEMS_IN_REG);
    }
  }
}
```

Code listing 2. Implementation of Maxloc using Intel AVX-512

Performance Evaluation

The Intel AVX-512 Maxloc implementation is compared to the baseline implementation shown in **Code listing 1** using the following experimental protocol:

- GCC (8.4.1) and Clang (11.0.0) are used with the `-O3 -march=icelake-server -mprefer-vector-width=512` flags to compile the baseline implementation. ICC (v19.1.3.304) is used to compile the AVX-512 implementation.
- The same input data is used in all benchmarks. The input data consists of values in ascending order (i.e., the maximum value is in the last index location). No performance differences were observed when random values were used.
- The input size is varied from 1,024 to 4,194,304 elements (16 MB with FP32 elements). This allows us to evaluate performance when the data fits into different cache hierarchies (L1D, L2, and L3).
- An Intel Xeon Platinum 8368 processor (3rd Gen Intel Xeon Scalable processor) is used for benchmarking. It has 38 cores per socket, 48 KB L1D cache, 1280 KB L2, and 57 MB L3 per socket.
- Performance is measured for one thread pinned to a single core. The average time elapsed for 100 iterations of each problem size is reported. Memory is allocated on 64B aligned boundaries.
- In the AVX-512 implementation, the main block is explicitly unrolled by 128 elements.

Table 2 shows the instruction sequence generated by GCC, Clang, and explicit Intel AVX-512 implementation compiled with ICC (main block only due to space constraints). **Figure 4** shows the performance of Maxloc with GCC, Clang, and explicit Intel AVX-512 methods. **Figure 5** shows the speed-up of the AVX-512 implementation over GCC. From the performance charts, we can make the following observations:

1. In the baseline implementation, performance is determined by the automatic vectorization ability of the compilers (GCC, Clang). As observed in the disassembly of object code generated in **Table 2**, neither GCC nor Clang is able to vectorize the computations, instead relying on scalar instructions (`vmovss, vucomiss, vmaxss, cmov`) that operate on a single FP32 element; hence, the "ss" suffix (stands for "single, single-precision") in the instruction name. Interestingly, even though Clang unrolls the loop by eight elements, it still processes the eight elements in serial order by emitting the same single, scalar instructions as GCC.
2. In contrast, the Intel AVX-512 implementation works on 16 FP32 elements stored in 512-bit wide ZMM registers using Intel AVX-512 instructions (as seen in the third column of **Table 2**). The Intel AVX-512 instructions map to the intrinsics in **Code listing 2**.
3. The Intel AVX-512 implementation of Maxloc delivers superior performance (**Figure 5**). The average speed-up for sizes that fit in L1, L2, and L3 caches are 18x, 40x, and 13x, respectively. As we hit L3 cache, the latency to load data into registers increases, so performance gains from Intel AVX-512 vectorization diminishes. For GCC and Clang, the lack of SIMD is reflected in the poor performance for all the problem sizes across caches. Even though Clang does unroll by eight elements, its performance is identical to GCC.

Baseline with GCC	Baseline with Clang	AVX512 with ICC
20: mov %rcx,%rdx 23: vmovss (%rsi,%rdx,4),%xmm1 28: lea 0x1(%rdx),%rcx 2c: vcomiss %xmm0,%xmm1 30: vmaxss %xmm0,%xmm1,%xmm0 34: cmova %edx,%eax 37: cmp %rdx,%rdi 3a: jne 20 <ref_max+0x20>	40: vmovss (%rsi,%rdx,4),%xmm1 45: vmovss 0x4(%rsi,%rdx,4),%xmm2 4b: vucomiss %xmm0,%xmm1 4f: cmova %edx,%eax 52: vmaxss %xmm0,%xmm1,%xmm0 56: lea 0x1(%rdx),%edi 59: vucomiss %xmm0,%xmm2 5d: cmovbe %eax,%edi 60: vmaxss %xmm0,%xmm2,%xmm0 64: vmovss 0x8(%rsi,%rdx,4),%xmm1 6a: lea 0x2(%rdx),%eax 6d: vucomiss %xmm0,%xmm1 71: cmovbe %edi,%eax 74: vmaxss %xmm0,%xmm1,%xmm0 78: vmovss 0xc(%rsi,%rdx,4),%xmm1 7e: lea 0x3(%rdx),%edi 81: vucomiss %xmm0,%xmm1 85: cmovbe %eax,%edi 88: vmovss 0x10(%rsi,%rdx,4),%xmm2 8e: vmaxss %xmm0,%xmm1,%xmm0 92: lea 0x4(%rdx),%eax 95: vucomiss %xmm0,%xmm2 99: cmovbe %edi,%eax 9c: vmaxss %xmm0,%xmm2,%xmm0 a0: vmovss 0x14(%rsi,%rdx,4),%xmm1 a6: lea 0x5(%rdx),%edi a9: vucomiss %xmm0,%xmm1 ad: cmovbe %eax,%edi b0: vmaxss %xmm0,%xmm1,%xmm0 b4: vmovss 0x18(%rsi,%rdx,4),%xmm1 ba: lea 0x6(%rdx),%ecx bd: vucomiss %xmm0,%xmm1 c1: cmovbe %edi,%ecx c4: vmaxss %xmm0,%xmm1,%xmm0 c8: vmovss 0x1c(%rsi,%rdx,4),%xmm1 ce: lea 0x7(%rdx),%eax d1: vucomiss %xmm0,%xmm1 d5: cmovbe %ecx,%eax d8: vmaxss %xmm0,%xmm1,%xmm0 dc: add $0x8,%rdx e0: cmp %rdx,%r8 e3: jne 40 <ref_max+0x40>	20: vmovups (%rsi),%zmm0 26: vmovups 0x80(%rsi),%zmm3 2d: vmovups 0x100(%rsi),%zmm4 34: vmovups 0x180(%rsi),%zmm5 3b: vmaxps 0x40(%rsi),%zmm0,%zmm6 42: vmaxps 0xc0(%rsi),%zmm3,%zmm7 49: vmaxps x140(%rsi),%zmm4,%zmm8 50: vmaxps x1c0(%rsi),%zmm5,%zmm9 57: vmaxps %zmm7,%zmm6,%zmm10 5d: vmaxps %zmm9,%zmm8,%zmm11 63: vmaxps zmm11,%zmm10,%zmm12 69: add $0x200,%rsi 70: vcmpgtps %zmm2,%zmm12,%k1 77: vpbroadcastd %eax,%zmm1{%k1} 7d: add $0x80,%eax 82: vblendmps %zmm12,%zmm2,%zmm2{%k1} 88: cmp %edx,%eax 8a: jl 20 <max_idx_tracking+0x20>

Table 2. Disassembly of Maxloc with GCC, Clang, and ICC

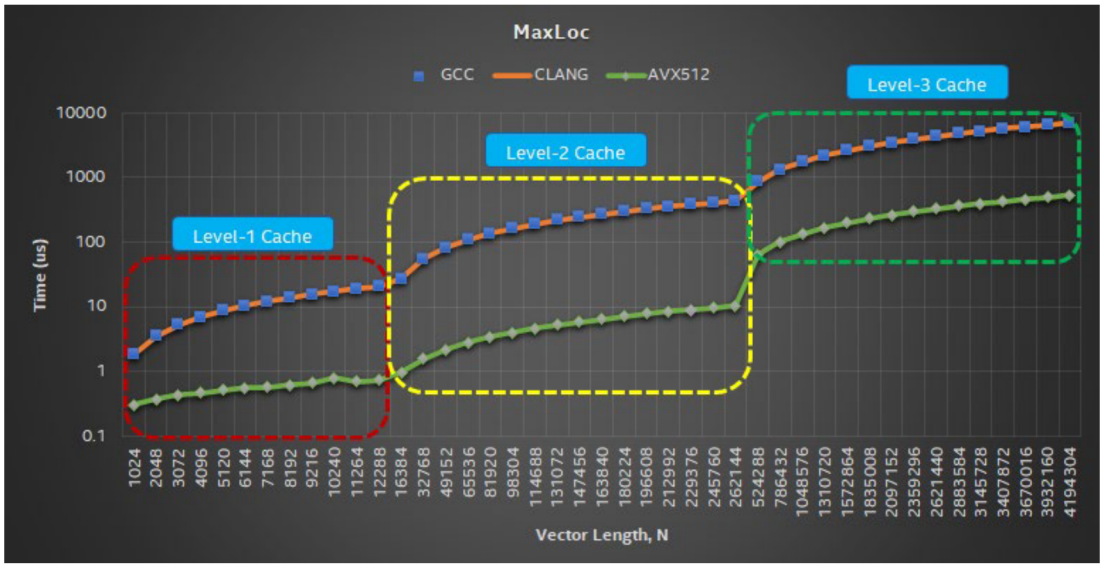

Figure 4. Performance of Maxloc with GCC, Clang, and ICC with explicit Intel AVX-512 vectorization

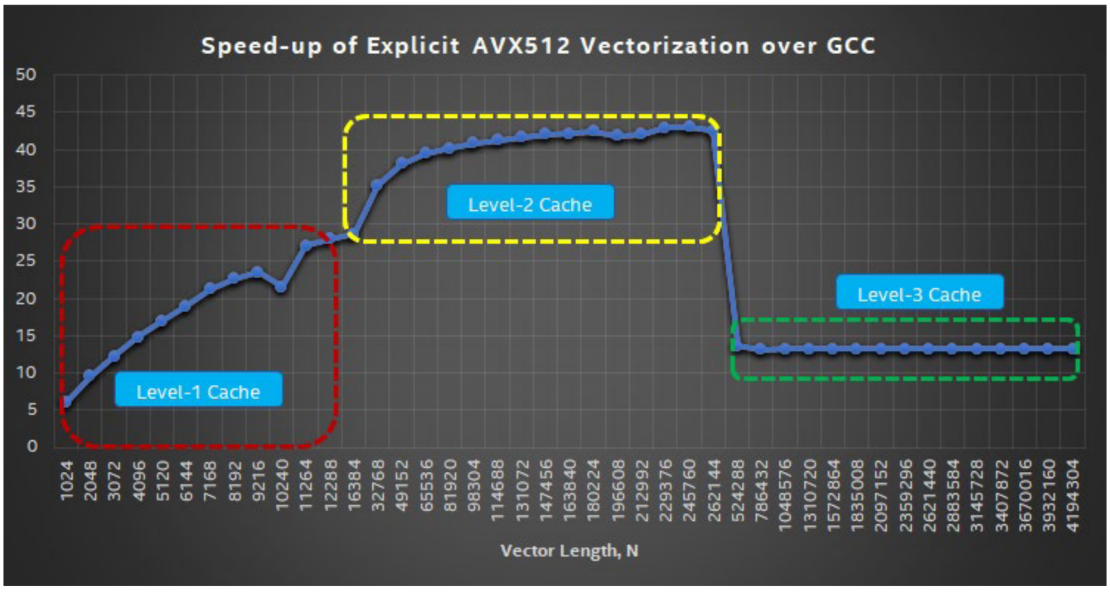

Figure 5. Speed-up of explicit AVX-512 vectorization over GCC

Conclusion

These results demonstrate the performance advantage of Intel AVX-512 instructions when applied to the Maxloc operation. Explicit AVX-512 vectorization delivers superior performance to GCC and Clang. I hope this demonstration will motivate you to explore Intel AVX-512 opportunities in your compute-intensive code.

intel software

THE PARALLEL UNIVERSE

Printed in USA 707/IH Please Recycle.