Fast Sorting Algorithms using AVX-512 on Intel Knights Landing

Berenger Bramas

Fast Sorting Algorithms using AVX-512 on Intel Knights Landing

Berenger Bramas

Max Planck Computing and Data Facility (MPCDF)
Email: Berenger.Bramas@mpcdf.mpg.de

This paper describes fast sorting techniques using the recent AVX-512 instruction set. Our implementations benefit from the latest possibilities offered by AVX-512 to vectorize a two-parts hybrid algorithm: we sort the small arrays using a branch-free Bitonic variant, and we provide a vectorized partitioning kernel which is the main component of the well-known Quicksort. Our algorithm sorts in-place and is straightforward to implement thanks to the new instructions. Meanwhile, we also show how an algorithm can be adapted and implemented with AVX-512. We report a performance study on the Intel KNL where our approach is faster than the GNU *C++* sort algorithm for any size in both integer and double floating-point arithmetics by a factor of 4 in average.

Keywords: quicksort; sort; vectorization; AVX-512; KNL

1. INTRODUCTION

Sorting is a fundamental problem in computer science, and efficient implementations are critical for various applications such as database servers [1] or image rendering engines [2]. Moreover, sorting libraries are massively used in software development to reduce the complexity of specific algorithms. As an example, once an array is sorted, it is possible to use binary search and find any item in logarithmic time. Therefore, having efficient sorting implementations on new architectures can improve performance of a wide range of applications.

From one CPU generation to the next, improvements and changes are made at various levels. Some modifications are hidden from the programmer and might improve existing codes without any update. This includes the low-level modules (speculation, out-of-order execution, etc.) but also the CPU's clock frequency. On the other hand, some new features and improvements require to be explicitly used. In this category, the two dominant improvements are the usage of vectorization units and multi-core parallelization. Thus, to achieve high-performance on modern CPUs, it is indispensable to *vectorize* a code, that is to take advantage of the single instruction on multiple data (SIMD) capacity. In fact, while the difference between a scalar code and its vectorized equivalent was *"only"* of a factor of 2 in the year 2000 (SSE technology and double precision), the difference is now up to a factor 4 on most CPUs (AVX) and up to 8 (AVX-512) on the KNL. Some algorithms or computational intensive kernels are straightforward to vectorize and could even be auto-vectorize by the compilers. However, data-processing algorithms are generally not in this category because the SIMD operations are not always well designed for this purpose.

The Intel Knights Landing (KNL) processor [3] did not follow the same path of improvement as the previous CPUs. In fact, the clock frequency has been reduced but the size of SIMD-vector and the number of cores in the chip have been increased impressively (in addition to having a new memory hierarchy). The KNL supports the AVX-512 instruction set [4]: it supports Intel AVX-512 foundational instructions (AVX-512F), Intel AVX-512 conflict detection instructions (AVX-512CD), Intel AVX-512 exponential and reciprocal instructions (AVX-512ER), and Intel AVX-512 prefetch instructions (AVX-512PF). AVX-512 allows to work on SIMD-vectors of double the size compared to previous AVX(2) set, and it comes with various new operations. Moreover, the next-generation of Intel CPUs will support AVX-512 too. Consequently, the development of new algorithms targeting Intel KNL should be beneficial to the future Intel Xeon SkyLake CPU as well.

In the current paper, we look at different strategies to develop an efficient sorting algorithm on Intel KNL using AVX-512. The contributions of this study are the following:

- studying sorting algorithms on the Intel KNL
- proposing a new partitioning algorithm using AVX-512
- defining a new Bitonic sort variant to sort tiny arrays using AVX-512
- implementing a new Quicksort variant using AVX-512.

All in all, we show how we can obtain a fast and

vectorized sorting algorithm [1].

The rest of the paper is organized as follows. Section 2 provides the background related to vectorization and sorting. Then, we describe our proposal to sort tiny arrays in Section 3, and our partitioning and Quicksort variant in Section 4. Finally, in Section 5, we provide a performance study of our method against the GNU *C++* standard sort (STL) implementation.

2. BACKGROUND

In this section, we provide the pre-requisite to our sort implementation. We first introduce the Quicksort and the Bitonic sort. Then we describe the objectives of the vectorization and we finish with an overview of some related work.

2.1. Sorting Algorithms

2.1.1. Quicksort Overview

We briefly describe the well-known Quicksort (QS) algorithm to make the document self-content, but readers comfortable with it can skip this section. QS has been originally proposed in [5], and uses a *divide-and-conquer* strategy by recursively partitioning the input array until it ends with partitions of one value. The partitioning puts values lower than a *pivot* at the beginning of the array and the values greater at the end with a linear complexity. QS has a worst-case complexity of $O(n^2)$ but an average complexity of $O(n \log n)$ in practice. The complexity is tied to the choice of the pivot when partitioning, it must be close to the median to ensure a low complexity. However, its simplicity in terms of implementation and its speed in practice has turned it into a very popular sorting algorithm. Figure 1 shows the sort of an array using QS.

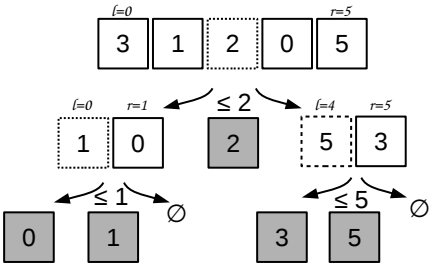


FIGURE 1: Quicksort example to sort $[3, 1, 2, 0, 5]$ to $[0, 1, 2, 3, 5]$. The pivot is equal to the value in the middle: the first pivot is 2, then at second recursion level it is 1 and 5.

We provide in Algorithm 2 a possible implementation of a scalar QS (here the term *scalar* refers to as single value at the opposite of a SIMD vector). In this implementation, the choice of the pivot is done naively by selecting the value in the middle before partitioning. This type of selection can result in very unbalanced partitions, hence why more advanced heuristics have been proposed in the past. As an example, the pivot can be selected by taking a median from several values.

Algorithm 1: Quicksort

```
   Input: array: an array to sort. length: the size of array.
   Output: array: the array sorted.
1  function QS(array, length)
2      QS_core(array, 0, length-1)
3  function QS_core(array, left, right)
4      if left < right then
5          // Naive method, select value in the middle
6          pivot_idx = ((right-left)/2) + left
7          swap(array[pivot_idx], array[right])
8          partition_bound = partition(array, left, right, array[right])
9          swap(array[partition_bound], array[right])
10         QS_core(array, left, partition_bound-1)
11         QS_core(array, partition_bound+1, right)
12     end
13 function partition(array, left, right, pivot_value)
14     for idx_read ← left to right do
15         if array[idx_read] ¡ pivot_value then
16             swap(array[idx_read], array[left])
17             left += 1
18         end
19     end
20     return left;
```

2.1.2. GNU std::sort Implementation (STL)

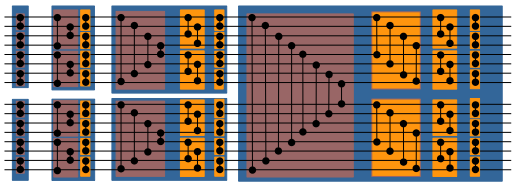
The worst case complexity of QS makes it no longer suitable to be used as the standard *C++* sort. In fact, a complexity of $O(n \log n)$ in average was required until year 2003 [6], but it is now a worst case limit [7] that a pure QS implementation cannot guarantee. Consequently, the current implementation is a 3-part hybrid sorting algorithm [8] i.e. it relies on 3 different algorithms. First, the algorithm uses an introsort [9] to a maximum depth of $2 \times log^2 n$ (introsort is a hybrid of quicksort and heap sort) to obtain small partitions that are then sorted using an insertion sort.

2.1.3. Bitonic Sorting Network

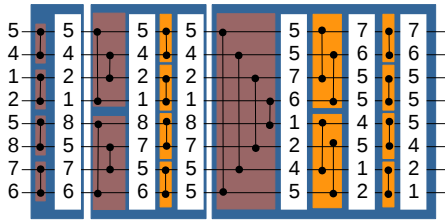
In computer science, a sorting network is an abstract description of how to sort values from a fixed-size array; how the values are compared and exchanged. A sorting network can be represented graphically having each input value as an horizontal lines and each *compare and exchange* unit as a vertical connection between those lines. It exists various sorting networks in the literature, but we concentrate our description on the Bitonic sort from [10]. This network is easy to implement and has an algorithm complexity of $O(n\,log(n)^2)$. Moreover, it has shown good performance on parallel computers [11] and GPUs [12]. Figure 2a shows a Bitonic sorting network to process 16 values. A sorting network can be seen as a time-line where input values are transfered from left to right and exchanged if needed at each vertical bar. We show such execution in Figure 2b where we print the

[1] The functions described in the current study are available at https://gitlab.mpcdf.mpg.de/bbramas/avx-512-sort. This repository includes a clean header-only library (branch master) and a test file that generates the performance study of the current manuscript (branch paper). The code is under MIT license.

intermediate steps while sorting an array of 8 values.



(a) Bitonic sorting network for input of size 16. All vertical bars/switches exchange values in the same direction.



(b) Example of 8 values sorted by a Bitonic sorting network.

FIGURE 2: Bitonic sorting network examples. In red boxes, the exchanges are done from extremities to the center. Whereas in orange boxes, the exchanges are done with a linear progression.

Two main strategies are possible when implementing a sorting network. Either it can be implemented by hard-coding the connections between the lines (which can be seen as a direct mapping of the picture). Or with a flexible algorithm that performs the same operations by using a formula/rule to decide when to exchange the values. In Algorithm 2, we show a flexible implementation of a Bitonic sort which mimics the networks presented in Figure 2a but without hard-coded exchanges.

2.2. Vectorization

The term vectorization refers to a CPUs' feature to apply a single operation/instruction to a vector of values instead of only a single one [13]. It is common to refer to this concept as Flynn's taxonomy term SIMD, for single instruction on multiple data. By adding SIMD instructions/registers to CPUs, it has been possible to increase the peak performance of the single cores despite the stagnation of the clock frequency since the mid 2000s. In the rest of the paper, we use the term SIMD-vector to call the data type managed by the CPU, but it has no relation with an expandable vector data structure such as *std::vector*. The size of a SIMD-vector is variable and depends on both the instruction set and the type of SIMD-vector element and is given by the size of the registers in the chip. Simd-vector extensions to the $X86$ instruction set, for example, are SSE [14], AVX [15], and AVX-512 [4], supporting SIMD-vectors of size 128, 256 and 512 bits, respectively. An AVX-512 SIMD-vector is able to store 8 double

Algorithm 2: Bitonic sort

```
   Input: array: an array to sort. length: the size of array (must be a
          power of 2).
   Output: array: the array sorted.
1  function bitonic_sort(array, length)
2      for s=2 ; s <= n ; s= s * 2 do
3          for i=0; i < n; i= i + s do
4              bitonic_core(sub_array(array,i), s);
5          end
6      end
7  function bitonic_core(array, n)
8      step = n/2
9      leftPtr = arr
10     rightPtr = sub_array(arr,n-1)
11     for k=0 ; k < step ; k= k + 1 do
12         test_and_exchange(leftPtr, rightPtr)
13         leftPtr = leftPtr + 1
14         rightPtr = rightPtr - 1
15     end
16     for step = n/2/2 ; step > 0 ; step = step/2 do
17         for i=0; i < n; i = i + step * 2 do
18             leftPtr = sub_array(arr,i)
19             rightPtr = sub_array(arr,i+step)
20             for k=0 ; k < step ; k= k + 1 do
21                 test_and_exchange(leftPtr, rightPtr)
22                 leftPtr = leftPtr + 1
23                 rightPtr = rightPtr + 1
24             end
25         end
26     end
```

precision floating-point numbers or 16 integer values, for example. Figure 3 illustrates the difference between a scalar summation and a SIMD-vector summation. Throughout this document, we use *intrinsic* function extension instead of the assembly language to write vectorized code on top of the AVX-512 instruction set. Intrinsics are small functions that should be replaced by the compiler with a single assembly instruction. Using intrinsics allows us to use high-level programming languages ($C++$ in our case) while being able to tell the compilers to use particular instructions.

FIGURE 3: Summation example of single precision floating-point values using : (■) scalar standard $C++$ code, (■) SSE SIMD-vector of 4 values , (■) AVX SIMD-vector of 8 values.

2.2.1. AVX-512

The AVX-512 is a recent instruction set that has been introduced with the Intel KNL CPU in the year 2016. It offers operations that do not have an equivalent in previous extensions of the $X86$ instruction sets. As an example, there are several new instructions that use a mask (integer) to select values or conditionally apply an operation on some values from a SIMD-vector. In the following, we describe some of the instructions that we use in our implementations.

Load/set/store. As in previous instruction sets, AVX-512 has instructions to load a contiguous block of values from main memory and to transform it into a SIMD-vector (load), fill a SIMD-vector with a given value (set) and move back a SIMD-vector into memory (store).

Store some. A new operation allows to save only some values from a SIMD-vector into memory (*vpcmpd/vcmppd*). The values are saved contiguously. This is a major improvement because without this instruction, several operations are needed to obtain the same result. For example, to save some values from a SIMD-vector v at address p in memory, one possibility is to load the current values from p into a SIMD-vector v', permute the values in v to move the values to store at the beginning, merge v and v', and finally save the resulting vector. The pseudo-code in Figure 4 describes the results obtained with this store-some instruction.

```
void _mm512_cmp_epi32_mask(
                    int* ptr,
                    __mask16 msk,
                    __m512i values){
    offset = 0;
    for( idx from 0 to 15){
        if(msk AND shift(1, idx)){
            ptr[offset] = values[idx];
            offset += 1;
        }
    }
}
```

Store some (integer)

```
void _mm512_cmp_pd_mask(
                    double* ptr,
                    __mask8 msk,
                    __m512d values){
    offset = 0;
    for( idx from 0 to 7){
        if(msk AND shift(1, idx)){
            ptr[offset] = values[idx];
            offset += 1;
        }
    }
}
```

Store some (double)

FIGURE 4: AVX-512 store-some behavior for an integer and a double floating-point vectors.

Vector permutation. Permuting the values inside a vector was possible since AVX/AVX2 using *permutevar*$8x32$ (instruction vperm(d,ps)). This instruction allows to re-order the values inside a SIMD-vector using a second integer array which contains the permutation indexes. AVX-512 also has this instruction on its own SIMD-vector, and we synthetize its behavior in Figure 5.

Min/Max. The min/max operations (*vpmaxsd/vpminsd/vmaxpd/vminpd*) return a SIMD-vector where each value correspond to the minimum/maximum value of the two input vectors at the same position (they do not return a single scalar as the global minimum/maximum among all the values). Such instructions exist in SSE/SSE2/AVX too.

```
__m512i _mm512_permutexvar_epi32(
                    __m512i permIdxs,
                    __m512i values){
    __m512i res;
    for( idx from 0 to 15)
        res[idx] = values[permIdxs[idx]];
    return res;
}
```

Permute (integer)

```
__m512d _mm512_permutexvar_pd(
                    __m512i permIdxs,
                    __m512d values){
    __m512d res;
    for( idx from 0 to 7)
        res[idx] = values[permIdxs[idx]];
    return res;
}
```

Permute (double)

FIGURE 5: AVX-512 permute behavior for an integer and a double floating-point vectors.

Comparison. In AVX-512, the value returned by a test/comparison (*vpcmpd/vcmppd*) is a mask (integer) and not a SIMD-vector of integers as it was in SSE/AVX. Therefore, it is easy to modify and work directly on the mask with arithmetic and binary operations for scalar integers. The behavior of the comparison is shown in Figure 6, where the mask is filled with bits from the comparison results. AVX-512 provides several instructions that use this type of mask like the conditional selection for instance.

Conditional selection. Among the mask-based instructions, the *mask move (vmovdqa32/vmovapd)* allows to select values between two vectors using a mask. The behavior is show in Figure 7, where a value is taken from the second vector where the mask is false and from the first vector otherwise. Achieving the same result was possible in previous instruction set only using several operations and not one dedicated instruction.

2.3. Vectorized Sorting Algorithms

The literature on sorting and vectorized sorting implementations is immense. Therefore, we only cite here some of the studies that we consider most connected to our work.

The sorting technique from [16] tries to remove branches and improves the prediction of a scalar sort, and they show a speedup by a factor of 2 against the STL (the implementation of the STL at that time was

```
__mask16 _mm512_cmp_epi32_mask(
                    __m512i values1,
                    __m512i values2,
                    operator op){
    __mask16 res = 0;
    for( idx from 0 to 15)
        if(op(values1[idx],values2[idx]) )
            res = res OR shift(1,idx);
    return res;
}
```

Comparison (integer)

```
__mask8 _mm512_cmp_pd_mask(
                    __m512d values1,
                    __m512d values2,
                    operator op){
    __mask8 res = 0;
    for( idx from 0 to 7)
        if(op(values1[idx],values2[idx]) )
            res = res OR shift(1,idx);
    return res;
}
```

Comparison (double)

FIGURE 6: AVX-512 comparison behavior for an integer and a double floating-point vectors.

```
__m512i _mm512_mask_mov_epi32(
                    __m512i sources,
                    __mask16 msk,
                    __m512i values){
    __m512i res;
    for( idx from 0 to 15){
        if(msk & shift(1, idx)){
            res[idx] = values[idx];
        }
        else{
            res[idx] = sources[idx];
        }
    }
}
```

Selection (integer)

```
__m512i _mm512_mask_mov_pd(
                    __m512d sources,
                    __mask8 msk,
                    __m512d values){
    __m512d res;
    for( idx from 0 to 7){
        if(msk & shift(1, idx)){
            res[idx] = values[idx];
        }
        else{
            res[idx] = sources[idx];
        }
    }
}
```

Selection (double)

FIGURE 7: AVX-512 selection behavior for an integer and a double floating-point vectors.

different). This study illustrates the early strategy to adapt sorting algorithms to a given hardware and shows the need of low-level optimizations due to the limited instructions available at that time.

In [17], the authors propose a parallel sorting on top of combosort vectorized with the VMX instruction set of IBM architecture. Unaligned memory accesses is avoided, and the L2 cache is efficiently managed by using a out-of-core/blocking scheme. The authors show a speedup by a factor of around 3 against the GNU *C++* STL.

In [18], the authors use a sorting-network for small-sized arrays similar to our approach. However, instead of dividing the main array into sorted partitions (partitions of increasing contents), and applying a small efficient sort on each of those partitions, the authors perform the contrary. Therefore, after multiple small sorts, the algorithms finishes with a complicated merge scheme using extra memory to globally sort all the partitions. A very similar approach has later been proposed in [19].

The recent work in [20] targets AVX2. The authors use a Quicksort variant with a vectorized partitioning and an insertion sort once the partitions are small enough (as the STL does). The partition method relies on a look-up table with a mapping between the comparison result of a SIMD-vector against the pivot, and the move/permutation that must be applied to the vector. The authors demonstrate a speedup by a factor of 4 against the STL. However, this method is not suitable on the KNL because the lookup table will need 256 values (leading to locality problems). This issue, as well as the use of extra memory, can be solved with the new instructions of the KNL. As a side remark, the authors do not compare their proposal to the standard *C++ partition* function even so it is the only part of their algorithm that is vectorized.

3. SORTING AVX-512 SIMD-VECTORS

In this section, we describe techniques to sort a small number of SIMD-vectors. As traditional sorting implementations usually rely on insertion sort to process small arrays, we propose here fully vectorized kernels that are later used in our final sorting implementation.

3.1. Naive Bubble Sort

3.1.1. Sorting one SIMD-vector

Using vectorization, we are able to work on all the values of a SIMD-vector without a loop statement. Therefore, while many basic sorting algorithms require two loops, one loop to reduce the number of values to sort while the second loop selects/moves the value to store at the end, vectorization allows to use only one. In Algorithm 3, we show how we implement a vectorized bubble sort on a single SIMD-vector. The idea is to exchange neighboring values (even/odd, odd/even) such that after a known number of iterations the full SIMD-vector is sorted. The number of iterations corresponds

to the length of the SIMD-vector VEC_SIZE divided by two (resulting in a total of VEC_SIZE exchanges). We provide in Appendix 1 the *C++* implementation of this algorithm, which is straightforward by using a combination of tests, permutations and selections. Figure 8 shows a diagram of the algorithm. A sorting-network equivalent to the diagram is obtained by simply unrolling the loop. In specific cases (when the data are more likely to be already sorted) it might beneficial to interrupt the loop if no values are exchanged by testing the comparison masks.

Algorithm 3: SIMD bubble sort for one vector.

```
   Input: vec: a AVX-512 vector to sort. VEC_SIZE the number of values
          in the vector (16 for vector of integers, 8 for vector of double
          floating-point values).
   Output: vec: the vector sorted.
1  function bubble_simd_sort(vec, VEC_SIZE)
2      for s=0 ; s < VEC_SIZE/2 ; s= s + 1 do
3          exchange_even_odd(vec)
4          exchange_odd_even(vec)
5          // Possible optimization, stop if nothing has been
               exchanged
6      end
```

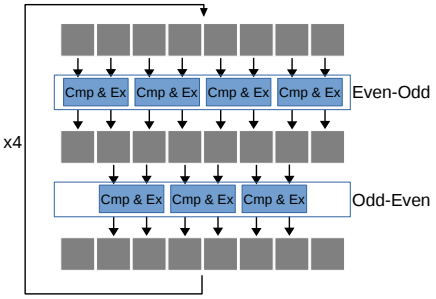


FIGURE 8: SIMD buble sort for one vector of double floating-points ($VEC\_SIZE = 8$).

3.1.2. Sorting several SIMD-vectors

The same strategy can be used for more than a one vector by propagating the values along all the vectors. We provide in Algorithm 4 the Bubble-sort for two SIMD-vectors where we exchange the border values between both vectors. The diagram of this algorithm is shown in Figure 9. This approach cannot be extended indefinitely because the number of SIMD registers is limited and because it has a quadratic complexity as the original bubble sort. However, for a small number of values, this method uses less instructions compared to its scalar equivalent and it can sort several vectors stored in registers without accessing the memory.

3.2. Bitonic-Based Sort

3.2.1. Sorting one SIMD-vector

Using the Bitonic sorting network to decide what values should be compared, instead of neighboring comparisons, allows us to reduce the number of instructions. We provide the pseudo-code to sort a single vector in Algorithm 5, where we use static (hard-coded) permutation vectors obtained from the Figure 2a. We obtain a fully vectorized branch-free implementation. The *C++* implementation of this function is given in Appendix 2 where *exchange_permute* is composed of a permutation, a min/max selection and a mask-based move.

Algorithm 4: SIMD bubble sort.

```
   Input: vec1 and vec2: two AVX-512 vectors to sort. VEC_SIZE the
          number of values in the vector (16 for vector of integers, 8 for
          vector of double floating-point values).
   Output: vec1 and vec2: the two vector sorted with vec1  vec2.
1  function bubble_simd_sort_2v(vec1, vec2, VEC_SIZE)
2      for s=0 ; s < VEC_SIZE ; s= s + 1 do
3          exchange_even_odd(vec1)
4          exchange_even_odd(vec2)
5          exchange_odd_even(vec1)
6          exchange_odd_even(vec2)
7          exchange_border(vec1,vec2)
8          // Possible optimization, stop if nothing has been
               exchanged
9      end
```

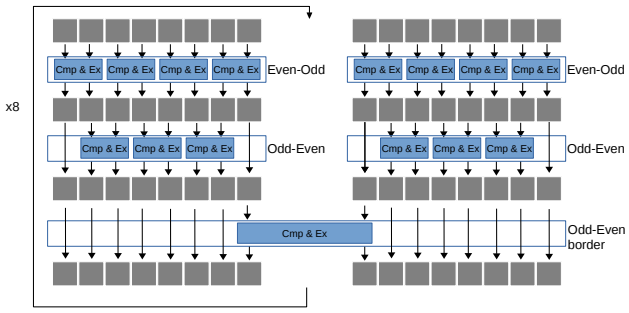


FIGURE 9: SIMD buble sort for two vectors of double floating points ($VEC\_SIZE = 8$).

Algorithm 5: SIMD bitonic sort for one vector of double floating-point values.

```
   Input: vec: a double floating-point AVX-512 vector to sort.
   Output: vec: the vector sorted.
1  function bitonic_simd_sort(vec)
2      exchange_permute(vec, [6, 7, 4, 5, 2, 3, 0, 1])
3      exchange_permute(vec, [4, 5, 6, 7, 0, 1, 2, 3])
4      exchange_permute(vec, [6, 7, 4, 5, 2, 3, 0, 1])
5      exchange_permute(vec, [0, 1, 2, 3, 4, 5, 6, 7])
6      exchange_permute(vec, [5, 4, 7, 6, 1, 0, 3, 2])
7      exchange_permute(vec, [6, 7, 4, 5, 2, 3, 0, 1])
```

3.2.2. Sorting several SIMD-vectors

Algorithm 7 sorts two SIMD-vectors using the Bitonic's comparison order. As shown in Figure 2a, we first have to apply the Bitonic sort to each vector individually before exchanging values between both vectors. The same principle can be applied to more vectors using the fact that to sort V vectors we re-use the function to sort $V/2$ vectors and so on.

Algorithm 6: SIMD bitonic sort for two vectors of double floating-point values.

```
   Input: vec1 and vec2: two double floating-point AVX-512 vectors to
          sort.
   Output: vec1 and vec2: the two vector sorted with vec1  vec2.
1  function bitonic_simd_sort_2v(vec1, vec2)
2      // Sort each vector using bitonic_simd_sort
3      bitonic_simd_sort(vec1)
4      bitonic_simd_sort(vec2)
5      exchange_permute(vec1, vec2, [0, 1, 2, 3, 4, 5, 6, 7])
6      exchange_permute(vec1, [3, 2, 1, 0, 7, 6, 5, 4])
7      exchange_permute(vec2, [3, 2, 1, 0, 7, 6, 5, 4])
8      exchange_permute(vec1, [5, 4, 7, 6, 1, 0, 3, 2])
9      exchange_permute(vec2, [5, 4, 7, 6, 1, 0, 3, 2])
10     exchange_permute(vec1, [6, 7, 4, 5, 2, 3, 0, 1])
11     exchange_permute(vec2, [6, 7, 4, 5, 2, 3, 0, 1])
```

3.3. Sorting Small Arrays

We describe in sections 3.1 and 3.2 how we can sort several SIMD-vectors. However, we intend to sort arrays with sizes not necessary multiple of the SIMD-vector's length. Therefore, we provide function that loads an array into SIMD-vectors, pads the last vector with extra-values and calls the appropriate vectorized sort. A possible implementation is shown in Algorithm 7 where we rely on a *switch* statement to select the existing function that matches the size of the array to sort.

Algorithm 7: SIMD bitonic-based sort for a small array (with a size less than $3 \times VEC\_SIZE$).

```
   Input: vec1 and vec2: two double floating-point AVX-512 vectors to
          sort.
   Output: vec1 and vec2: the two vector sorted with vec1  vec2.
1  function simd_sort_up_to_3v(array, length)
2      nb_vecs = (length+VEC_SIZE-1)/VEC_SIZE
3      padding = nb_vecs*VEC_SIZE-length
4      last_vec_size = VEC_SIZE-padding
5      switch nb_vecs do
6          case 0
7              v1 = simd_load_some(array, 0, last_vec_size)
8              v1 = simd_padd_vec(v1, last_vec_size, MAXIMUM)
9              bitonic_simd_sort(v1)
10             simd_store_some(array, 0, last_vec_size, v1)
11         end
12         case 1
13             v1 = simd_load(array, 0)
14             v2 = simd_load_some(array, VEC_SIZE,
               last_vec_size)
15             v2 = simd_padd_vec(v2, last_vec_size, MAXIMUM)
16             bitonic_simd_sort_2v(v1,v2)
17             simd_store(array, 0, v1)
18             simd_store_some(array, 0, last_vec_size, v2)
19         end
20         otherwise
21             v1 = simd_load(array, 0)
22             v2 = simd_load(array, VEC_SIZE)
23             v3 = simd_load_some(array, VEC_SIZE × 2,
               last_vec_size)
24             v3 = simd_padd_vec(v3, last_vec_size, MAXIMUM)
25             bitonic_simd_sort_3v(v1,v2,v3)
26             simd_store(array, 0, v1)
27             simd_store(array, VEC_SIZE, v2)
28             simd_store_some(array, VEC_SIZE × 2,
               last_vec_size, v3)
29         end
30     endsw
```

4. HIGH LEVEL OPERATIONS

4.1. Partitioning with AVX-512

The store-some instruction from AVX-512 (described in Section 2.2.1) is a key operation of our partitioning function. It allows us to store the values from a SIMD-vector lower/greater than a pivot directly in memory. The main idea of our algorithm is to load values from the array and directly store the results using two iterators on the left and right for the values lower and greater than the pivot, respectively. To avoid overwriting values that have not been proceed, our algorithm starts by loading two SIMD-vectors (one from each array's extremities). Consequently, our implementation works in-place and only needs three SIMD-vectors.

We show our method in Algorithm 8. This description also include as side comments some possible optimizations in case the array is more likely to be already partitioned (A) or to reduce the data displacement of the values (B). We provide the AVX-512 implementation of this function in Appendix 3.

4.2. Quicksort Variant

In Section 4.1 we introduced a vectorized partitioning algorithm, and in Section 3.3 we described a vectorized sort function for small arrays. As a result, we transform the original QS (Algorithm 2) into Algorithm 9 where we replace the scalar partitioning by our simd-partitioning method, and where we sort small partitions using our Bitonic-based sort. The worst-case complexity of this SIMD-QS is $O(n^2)$, as the sequential QS, and tied to the choice of the pivot.

5. PERFORMANCE STUDY

5.1. Configuration

We asses our method on an Intel Xeon Phi CPU (KNL) 7210 at 1.30GHz. The cluster memory mode has been set to Quadrant/Flat, and we bind the process and the allocation with *numactl –physcpubind=0 –membind=1.* We use GCC 6.2.0, but we provide the results of the same tests with Intel compiler 17.0.1 in Appendix A.1 (both compilers deliver the same performance). The test file used for the presented benchmark is available online[2], it includes the different sorts presented in this study plus some additional strategies and tests. Our SIMD-QS uses a 3-values median pivot selection (similar to the STL sort function).

5.2. Sorting AVX-512 Vectors

5.2.1. Fixed Size

In Figure 10, we compare the time to sort from 1 to 4 SIMD-vectors. AVX-512-bubble provides an important

[2]The test file that generates the performance study is available at https://gitlab.mpcdf.mpg.de/bbramas/avx-512-sort (branch paper) under MIT license.

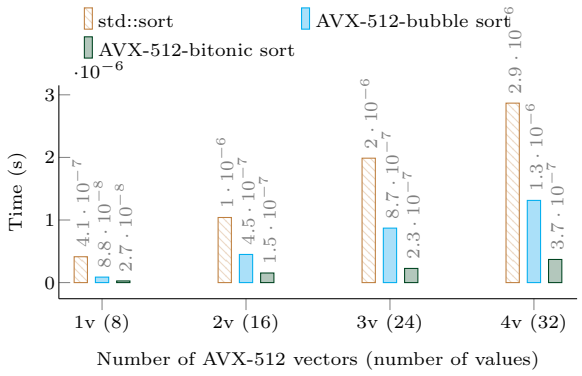
Algorithm 8: SIMD partitioning. VEC_SIZE is the number of values inside a SIMD-vector of type array's elements.

```
   Input: array: an array to parition. length: the size of array. pivot: the
          reference value
   Output: array: the array parttitioned. left_w: the index between the
           values lower and larger than the pivot.
1  function SIMD_partition(array, length, pivot)
2      // If too small use scalar partitioning
3      if length ≤ 2 × VEC_SIZE then
4          Scalar_partition(array, length)
5          return
6      end
7      // Fill a vector with all values set to pivot
8      pivotvec = simd_set_from_one(pivot);
9      // Init iterators and save one value on each extremity
10     left = 0
11     left_w = 0
12     left_vec = simd_load(array, left)
13     left = left + VEC_SIZE
14     right = length-VEC_SIZE
15     right_w = length
16     right_vec = simd_load(array, right)
17     while left + VEC_SIZE ≤ right do
18         if (left - left_w) ≤ (right_w - right) then
19             val = simd_load(array, left)
20             left = left + VEC_SIZE
21             // (B) Possible optimization, swap val and
                   left_vec
22         else
23             right = right - VEC_SIZE
24             val = simd_load(array, right)
25             // (B) Possible optimization, swap val and
                   right_vec
26         end
27         [left_w, right_w] = proceed(array, val, pivotvec, left_w,
           right_w)
28     end
29     // Process left_val and right_val
30     [left_w, right_w] = proceed(array, left_val, pivotvec, left_w,
       right_w)
31     [left_w, right_w] = proceed(array, right_val, pivotvec, left_w,
       right_w)
32     // Proceed remaining values (less than VEC_SIZE
          values)
33     nb_remaining = right - left
34     val = simd_load(array, left)
35     left = right
36     mask = get_mask_less_equal(val, pivotvec)
37     mask_low = cut_mask(mask, nb_remaining)
38     mask_high = cut_mask(reverse_mask(mask) , nb_remaining)
39     // (A) Possible optimization, do only if mask_low is not
          0
40     simd_store_some(array, left_w, mask_low, val)
41     left_w = left_w + mask_nb_true(mask_low)
42     // (A) Possible optimization, do only if mask_high is not
          0
43     right_w = right_w - mask_nb_true(mask_high)
44     simd_store_some(array, right_w, mask_high, val)
45     return left_w;
46 function proceed(array, val, pivotvec, left_w, right_w)
47     mask = get_mask_less_equal(val, pivotvec);
48     nb_low = mask_nb_true(mask)
49     nb_high = VEC_SIZE-nb_low
50     // (A) Possible optimization, do only if mask is not 0
51     simd_store_some(array, left_w, mask, val)
52     left_w = left_w + nb_low
53     // (A) Possible optimization, do only if mask is not all
          true
54     right_w = right_w - nb_high
55     simd_store_some(array, right_w, reverse_mask(mask), val)
56     return [left_w, right_w]
```

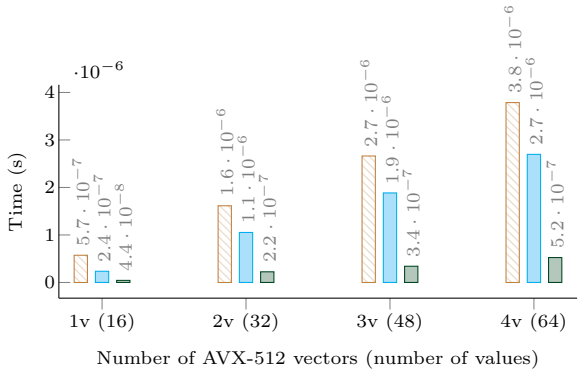
Algorithm 9: SIMD Quicksort. *select_pivot_pos* returns a pivot.

```
   Input: array: an array to sort. length: the size of array.
   Output: array: the array sorted.
1  function SIMD_QS(array, length)
2      SIMD_QS_core(array, 0, length-1)
3  function SIMD_QS_core(array, left, right)
4      if left + SORT_BOUND < right then
5          pivot_idx = select_pivot_pos(array, left, right)
6          swap(array[pivot_idx], array[right])
7          partition_bound = SIMD_partition(array, left, right,
           array[right])
8          swap(array[partition_bound], array[right])
9          SIMD_QS_core(array, left, partition_bound-1)
10         SIMD_QS_core(array, partition_bound+1, right)
11     else
12         // Could be simd_sort_up_to_X
13         SIMD_small_sort(sub_array(array, left), right-left+1)
14     end
```

speedup compared to the STL, but the AVX-512-bitonic is even faster showing a speedup factor up to 10 against the STL. Both tested arithmetics have a similar behavior even though there are 16 values inside an integer SIMD-vector and 8 in a double floating-point SIMD-vector. For the same number of values, the AVX-512-bitonic appears slightly faster to sort integer values compared to floating-point ones. This is not surprising since integers are smaller and comparing integers is faster than comparing floating-point numbers. For the rest of the study, we leave the bubble-sort aside because it is slower than the bitonic-based sort.

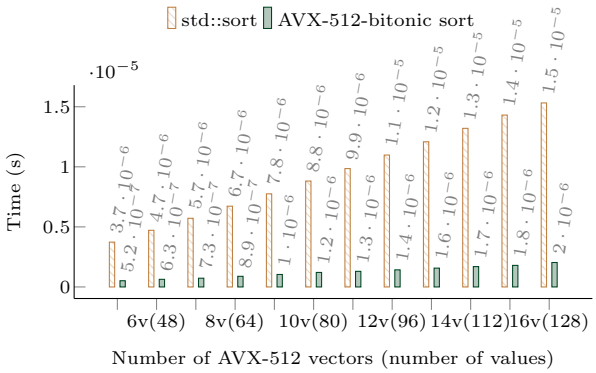


(a) Floating-point (double)

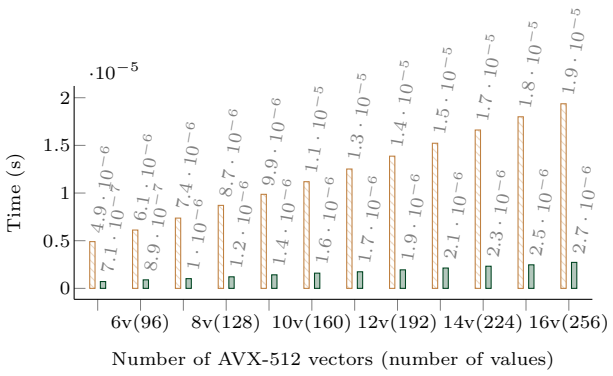


(b) Integer (int)

FIGURE 10: Sorting from 1 to 4 AVX-512 vectors of double and int (GCC compiler). The arrays are randomly generated and the execution time is obtained from the average of $\approx 10^7$ calls. Execution times are shown above each bar.



(a) Floating-point (double)



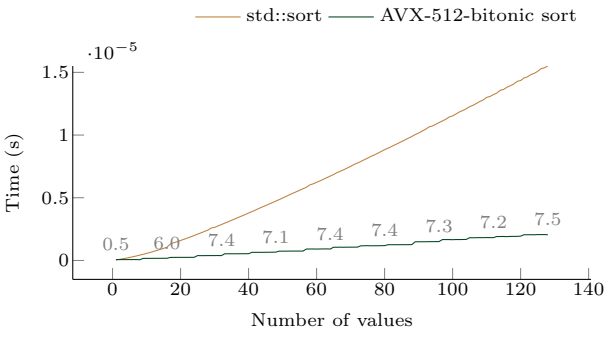
(b) Integer (int)

FIGURE 11: Sorting from 5 to 16 AVX-512 vectors of double and int (GCC compiler). The arrays are randomly generated and the execution time is obtained from the average of $\approx 10^7$ calls. Execution times are shown above each bar.

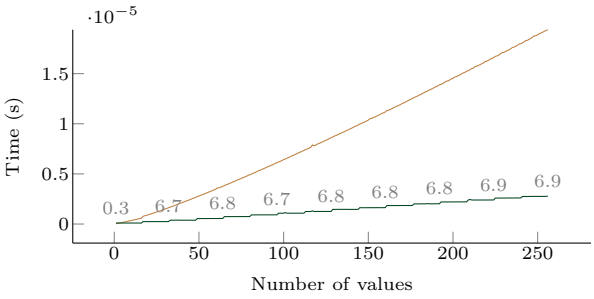
In Figure 11, we compare the time to sort up to 16 SIMD-vectors. The plot shows that the AVX-512-bitonic sort maintains its speedup against the STL. However, our implementation use hard coded permutation indices, so we cannot extend the size of the array to sort indefinitely. Moreover, the arrays we sort in this figure have size multiple of the SIMD-vector length, but we intend to sort any size.

5.2.2. Variable Size

Figure 12 shows the execution time to sort arrays of size from 1 to $16 \times VEC\_SIZE$ including those of size not multiple of the length of SIMD-vectors. We remind that we use the method from Section 3.3 to pad the last SIMD-vectors with extra values. We see that AVX-512-bitonic still delivers a good speedup against the STL. We can observe that the execution time increases every VEC_SIZE values because the cost of sorting is not tied to the number of values but to the number of SIMD-vectors to sort.



(a) Floating-point (double)



(b) Integer (int)

FIGURE 12: Execution time to sort from 1 to $16 \times VEC\_SIZE$ values of double and int (GCC compiler). The execution time is obtained from the average of 10^4 sorts for each size.The speedup of the AVX-512-bitonic against the STL is shown above the AVX-512-bitonic line.

5.3. AVX-512-Partition

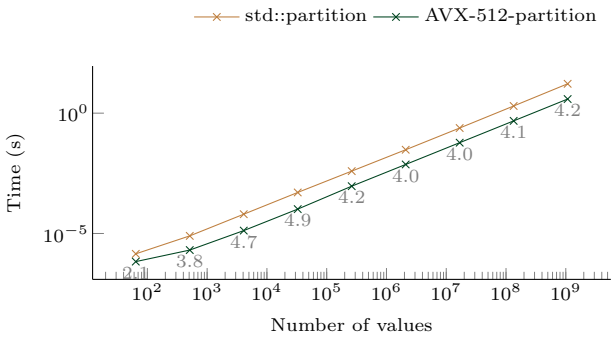
Figure 13 shows the execution time to partition using the *std::partition*, our AVX-512-partition and a basic partition implementation. Our AVX-512-partition is faster for any size, whereas both other methods appear quite similar in terms of performance. This speedup was not easy to predict. In fact, while the number of instructions of the vectorized implementation is smaller than its scalar equivalent, the partition operation is memory bound.

5.4. AVX-512-QS Sort

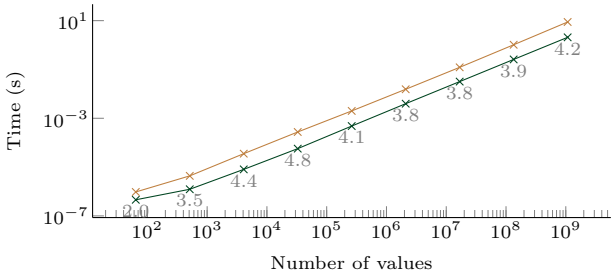
The execution time of our SIMD-QS against the STL is shown in Figure 14. The speedup of our implementation is still significant but not as high as for the partitioning. In addition, it appears that the difference decreases as the size of the array increases, showing the STL gain by using a 3-parts sort with a complexity guarantee. However, our implementation still provides a significant speedup even when sorting more than $8G$ bytes of values.

6. CONCLUSIONS

In this paper, we have described new sorting strategies using the AVX-512 instruction set on the Intel KNL.

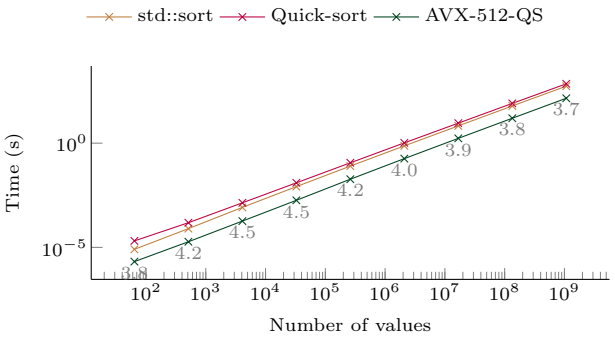


(a) Floating-point (double)

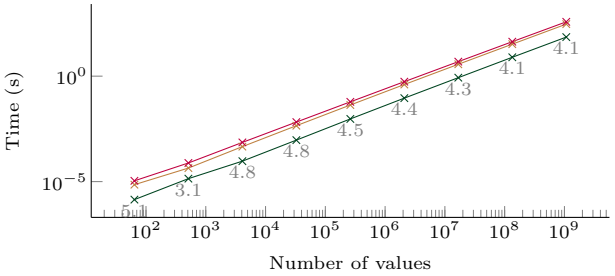


(b) Integer (int)

FIGURE 13: Execution time to partition arrays filled with random values with sizes from 2^1 to $2^{30}(\approx 10^9)$ of double and int (GCC compiler).The pivot is selected randomly. The speedup of the AVX-512-partition against the STL is shown above the AVX-512-partition line. The execution time is obtained from the average of 20 executions.



(a) Floating-point (double)



(b) Integer (int)

FIGURE 14: Execution time to sort arrays filled with random values with sizes from 2^1 to $2^{30}(\approx 10^9)$ of double and int (GCC compiler). The speedup of the AVX-512-QS against the STL is shown above the AVX-512-QS line. The execution time is obtained from the average of 5 executions.

The AVX-512 instruction set provides unique SIMD operations helpful to vectorize and to adapt algorithms more naturally than with SSE or AVX(2). We have introduced two approaches to sort small arrays: one based on the Bubble sort and another on the Bitonic sorting network. Both techniques are fully vectorized and provide a significant speedup against the STL, even for sorting arrays with sizes not multiple of the SIMD-vector's length. We have introduced a AVX-512 partitioning function, which is fully vectorized in its steady state and partitions in-place. Finally, our main sorting algorithm is a 2-part sort which partitions the input array until the partitions are small enough to be sorted by the Bitonic-based function (less than 16 SIMD-vectors length). We study the performance of each layer with different granularities on an Intel KNL. Our implementations show superior performance against the STL to partition or to sort arrays of any size. In the future, we intend to design a parallel implementation of our AVX-512-QS to use the 64 cores that the KNL provides, and we expect the recursive partitioning to be naturally parallelized with a task-based scheme on top of OpenMP. Additionally, we will evaluate our implementations on the next Intel Xeon SKYLAKE CPU.

REFERENCES

[1] Graefe, G. (2006) Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, **38**, 10.

[2] Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M. (1998) Designing a pc game engine. *IEEE Computer Graphics and Applications*, **18**, 46–53.

[3] Intel. Intel xeon phi processors. http://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html. Accessed: January 2017.

[4] Intel. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf. Accessed: December 2016.

[5] Hoare, C. A. (1962) Quicksort. *The Computer Journal*, **5**, 10–16.

[6] (2003). Iso/iec 14882:2003(e): Programming languages - c++. 25.3.1.1 sort [lib.sort] para. 2.

[7] (2014-11-19). Working draft, standard for programming language c++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf. 25.4.1.1 sort (p. 911).

[8] libstdc++ documentation: Sorting algorithm. https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html#l05207.

[9] Musser, D. R. (1997) Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, **27**, 983–993.

[10] Batcher, K. E. (1968) Sorting networks and their applications. *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 307–314. ACM.

[11] Nassimi, D. and Sahni, S. (1979) Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Computers*, **28**, 2–7.

[12] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008) Gpu computing. *Proceedings of the IEEE*, **96**, 879–899.

[13] Kogge, P. M. (1981) *The architecture of pipelined computers*. CRC Press.

[14] Intel. Intel 64 and ia-32 architectures software developer's manual: Instruction set reference (2a, 2b, 2c, and 2d). https://software.intel.com/en-us/articles/intel-sdm. Accessed: December 2016.

[15] Intel. Introduction to intel advanced vector extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions. Accessed: December 2016.

[16] Sanders, P. and Winkel, S. (2004) Super scalar sample sort. *European Symposium on Algorithms*, pp. 784–796. Springer.

[17] Inoue, H., Moriyama, T., Komatsu, H., and Nakatani, T. (2007) Aa-sort: A new parallel sorting algorithm for multi-core simd processors. *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 189–198. IEEE Computer Society.

[18] Furtak, T., Amaral, J. N., and Niewiadomski, R. (2007) Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 348–357. ACM.

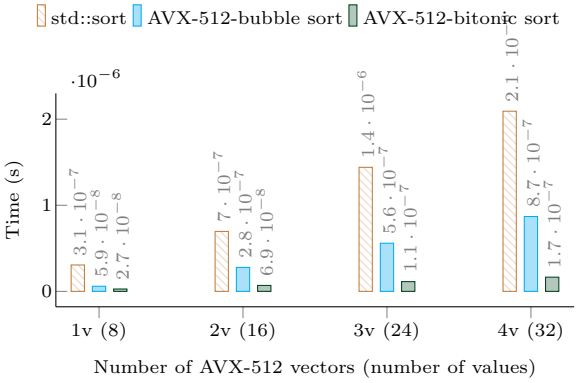
[19] Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar, S., and Dubey, P. (2008) Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, **1**, 1313–1324.

[20] Gueron, S. and Krasnov, V. (2016) Fast quicksort implementation using avx instructions. *The Computer Journal*, **59**, 83–90.

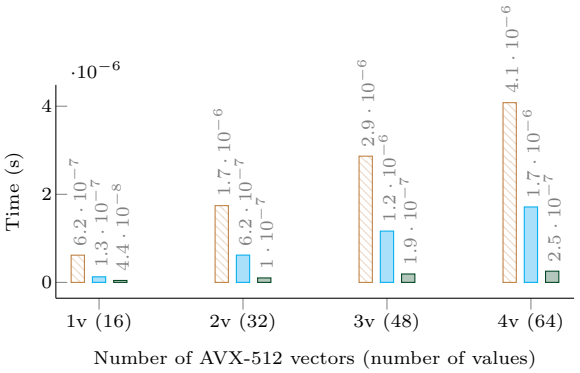
[21] IBM. Performance optimization and tuning techniques for ibm power systems processors including ibm power8. https://www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf. Accessed: December 2016.

APPENDIX

Appendix A.1. Performance Results (Intel Compiler)

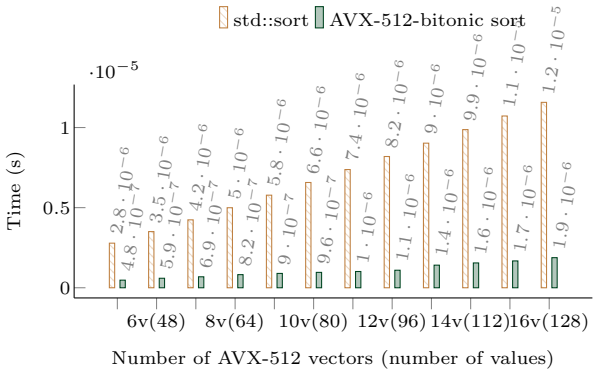


(a) Floating-point (double)

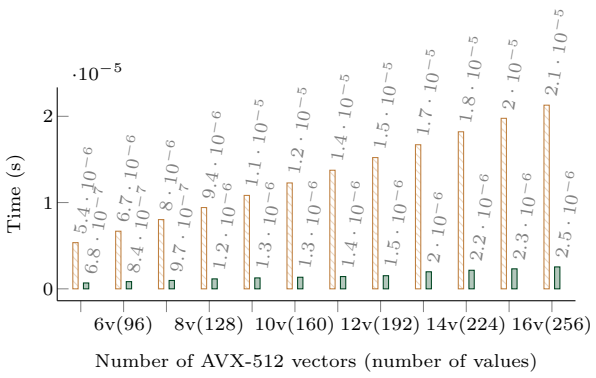


(b) Integer (int)

FIGURE A.1: Sorting from 1 to 4 AVX-512 vectors of double and int (Intel compiler). The arrays are randomly generated and the execution time is obtained from the average of $\approx 10^7$ calls. Execution times are shown above each bar.

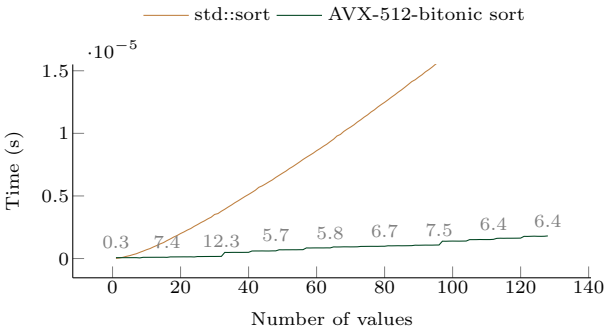


(a) Floating-point (double)

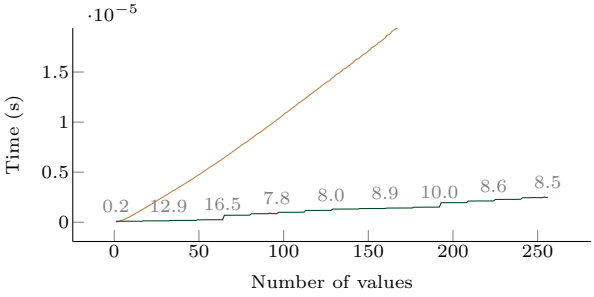


(b) Integer (int)

FIGURE A.2: Sorting from 5 to 16 AVX-512 vectors of double and int (Intel compiler). The arrays are randomly generated and the execution time is obtained from the average of $\approx 10^7$ calls. Execution times are shown above each bar.

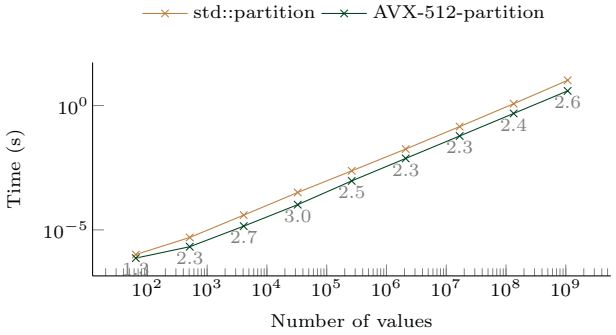


(a) Floating-point (double)

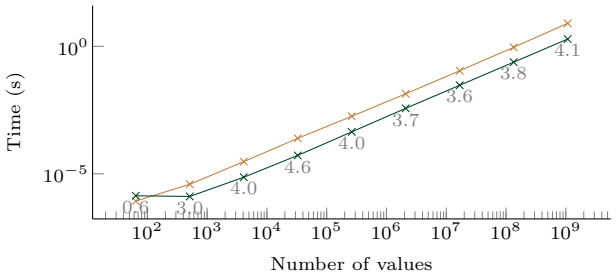


(b) Integer (int)

FIGURE A.3: Execution time to sort from 1 to 16 $\times$ VEC_SIZE values of double and int (Intel compiler). The execution time is obtained from the average of 10^4 sorts for each size.The speedup of the AVX-512-bitonic against the STL is shown above the AVX-512-bitonic line.

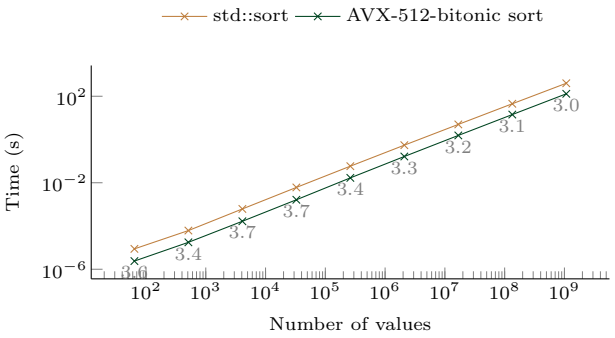


(a) Floating-point (double)

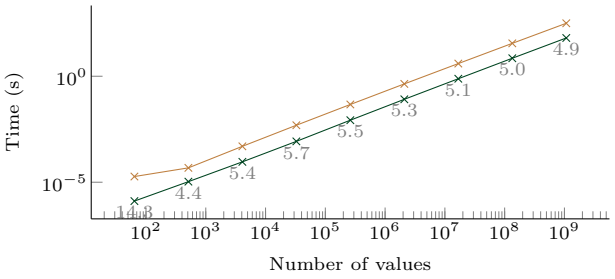


(b) Integer (int)

FIGURE A.4: Execution time to partition arrays filled with random values with sizes from 2^1 to $2^{30}(\approx 10^9)$ of double and int (Intel compiler).The pivot is selected randomly. The speedup of the AVX-512-partition against the STL is shown above the AVX-512-partition line. The execution time is obtained from the average of 20 executions.



(a) Floating-point (double)



(b) Integer (int)

FIGURE A.5: Execution time to sort arrays filled with random values with sizes from 2^1 to $2^{30}(\approx 10^9)$ of double and int (Intel compiler). The speedup of the AVX-512-QS against the STL is shown above the AVX-512-QS line. The execution time is obtained from the average of 5 executions.

Source Code Extracts

```
inline __m512d SortVec(__m512d input){
    __m512i idxNoNeigh = _mm512_set_epi64(7, 5, 6, 3, 4, 1, 2, 0);
    for( int idx = 0 ; idx < 4 ; ++idx){
        __m512d permNeighOdd = _mm512_permute_pd(input, 0x55);
        __mmask8 compMaskOdd = _mm512_cmp_pd_mask(permNeighOdd, input, _CMP_LT_OQ);
        input = _mm512_mask_mov_pd(input, (compMaskOdd & 0x55) | ((compMaskOdd & 0x55)<<1), ...
    permNeighOdd);

        __m512d permNeighEven = _mm512_permutexvar_pd(idxNoNeigh, input);
        __mmask8 compMaskEven = _mm512_cmp_pd_mask(permNeighEven, input, _CMP_LT_OQ);
        input = _mm512_mask_mov_pd(input, (compMaskEven & 0x2A) | ((compMaskEven & 0x2A)<<1), ...
    permNeighEven);
    }
    return input;
}

```

Code 1: AVX-512 Bubble sort for one simd-vector of double floating-point values.

```
inline __m512d CoreSmallSort(__m512d input){
    {
        __m512i idxNoNeigh = _mm512_set_epi64(6, 7, 4, 5, 2, 3, 0, 1);
        __m512d permNeigh = _mm512_permutexvar_pd(idxNoNeigh, input);
        __m512d permNeighMin = _mm512_min_pd(permNeigh, input);
        __m512d permNeighMax = _mm512_max_pd(permNeigh, input);
        input = _mm512_mask_mov_pd(permNeighMin, 0xAA, permNeighMax);
    }
    {
        __m512i idxNoNeigh = _mm512_set_epi64(4, 5, 6, 7, 0, 1, 2, 3);
        __m512d permNeigh = _mm512_permutexvar_pd(idxNoNeigh, input);
        __m512d permNeighMin = _mm512_min_pd(permNeigh, input);
        __m512d permNeighMax = _mm512_max_pd(permNeigh, input);
        input = _mm512_mask_mov_pd(permNeighMin, 0xCC, permNeighMax);
    }
    {
        __m512i idxNoNeigh = _mm512_set_epi64(6, 7, 4, 5, 2, 3, 0, 1);
        __m512d permNeigh = _mm512_permutexvar_pd(idxNoNeigh, input);
        __m512d permNeighMin = _mm512_min_pd(permNeigh, input);
        __m512d permNeighMax = _mm512_max_pd(permNeigh, input);
        input = _mm512_mask_mov_pd(permNeighMin, 0xAA, permNeighMax);
    }
    {
        __m512i idxNoNeigh = _mm512_set_epi64(0, 1, 2, 3, 4, 5, 6, 7);
        __m512d permNeigh = _mm512_permutexvar_pd(idxNoNeigh, input);
        __m512d permNeighMin = _mm512_min_pd(permNeigh, input);
        __m512d permNeighMax = _mm512_max_pd(permNeigh, input);
        input = _mm512_mask_mov_pd(permNeighMin, 0xF0, permNeighMax);
    }
    {
        __m512i idxNoNeigh = _mm512_set_epi64(5, 4, 7, 6, 1, 0, 3, 2);
        __m512d permNeigh = _mm512_permutexvar_pd(idxNoNeigh, input);
        __m512d permNeighMin = _mm512_min_pd(permNeigh, input);
        __m512d permNeighMax = _mm512_max_pd(permNeigh, input);
        input = _mm512_mask_mov_pd(permNeighMin, 0xCC, permNeighMax);
    }
    {
        __m512i idxNoNeigh = _mm512_set_epi64(6, 7, 4, 5, 2, 3, 0, 1);
        __m512d permNeigh = _mm512_permutexvar_pd(idxNoNeigh, input);
        __m512d permNeighMin = _mm512_min_pd(permNeigh, input);
        __m512d permNeighMax = _mm512_max_pd(permNeigh, input);
        input = _mm512_mask_mov_pd(permNeighMin, 0xAA, permNeighMax);
    }

    return input;
}

```

Code 2: AVX-512 Bitonic sort for one simd-vector of double floating-point values.

```
template <class IndexType>
static inline IndexType Partition512(double array[], IndexType left, IndexType right,
                                     const double pivot){
    const IndexType S = 8;//(512/8)/sizeof(double);

    if(right-left+1 < 2*S){
        return CoreScalarPartition<double,IndexType>(array, left, right, pivot);
    }

    __m512d pivotvec = _mm512_set1_pd(pivot);

    __m512d left_val = _mm512_loadu_pd(&array[left]);
    IndexType left_w = left;
    left += S;

    IndexType right_w = right+1;
    right -= S-1;
    __m512d right_val = _mm512_loadu_pd(&array[right]);

    while(left + S <= right){
        const IndexType free_left = left - left_w;
        const IndexType free_right = right_w - right;

        __m512d val;
        if( free_left <= free_right ){
            val = _mm512_loadu_pd(&array[left]);
            left += S;
        }
        else{
            right -= S;
            val = _mm512_loadu_pd(&array[right]);
        }

        __mmask8 mask = _mm512_cmp_pd_mask(val, pivotvec, _CMP_LE_OQ);

        const IndexType nb_low = popcount(mask);
        const IndexType nb_high = S-nb_low;

        _mm512_mask_compressstoreu_pd(&array[left_w],mask,val);
        left_w += nb_low;

        right_w -= nb_high;
        _mm512_mask_compressstoreu_pd(&array[right_w],~mask,val);
    }

    {
        const IndexType remaining = right - left;
        __m512d val = _mm512_loadu_pd(&array[left]);
        left = right;

        __mmask8 mask = _mm512_cmp_pd_mask(val, pivotvec, _CMP_LE_OQ);

        __mmask8 mask_low = mask & ~(0xFF << remaining);
        __mmask8 mask_high = (~mask) & ~(0xFF << remaining);

        const IndexType nb_low = popcount(mask_low);
        const IndexType nb_high = popcount(mask_high);

        _mm512_mask_compressstoreu_pd(&array[left_w],mask_low,val);
        left_w += nb_low;

        right_w -= nb_high;
        _mm512_mask_compressstoreu_pd(&array[right_w],mask_high,val);
    }
    {
        __mmask8 mask = _mm512_cmp_pd_mask(left_val, pivotvec, _CMP_LE_OQ);

        const IndexType nb_low = popcount(mask);
        const IndexType nb_high = S-nb_low;

        _mm512_mask_compressstoreu_pd(&array[left_w],mask,left_val);
        left_w += nb_low;

        right_w -= nb_high;
        _mm512_mask_compressstoreu_pd(&array[right_w],~mask,left_val);
    }

```

```
77
78      {
79          __mmask8 mask = _mm512_cmp_pd_mask(right_val, pivotvec, _CMP_LE_OQ);
80
81          const IndexType nb_low = popcount(mask);
82          const IndexType nb_high = S-nb_low;
83
84          _mm512_mask_compressstoreu_pd(&array[left_w],mask,right_val);
85          left_w += nb_low;
86
87          right_w -= nb_high;
88          _mm512_mask_compressstoreu_pd(&array[right_w],~mask,right_val);
89      }
90      return left_w;
91  }
92
```

Code 3: AVX-512 partitioning of a double floating-point array