A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake

Berenger Bramas

A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake

Berenger Bramas
Max Planck Computing and Data Facility (MPCDF)
Gieenbachstrae 2
85748 Garching, Germany

***Abstract*—The modern CPU's design, which is composed of hierarchical memory and SIMD/vectorization capability, governs the potential for algorithms to be transformed into efficient implementations. The release of the AVX-512 changed things radically, and motivated us to search for an efficient sorting algorithm that can take advantage of it. In this paper, we describe the best strategy we have found, which is a novel two parts hybrid sort, based on the well-known Quicksort algorithm. The central partitioning operation is performed by a new algorithm, and small partitions/arrays are sorted using a branch-free Bitonic-based sort. This study is also an illustration of how classical algorithms can be adapted and enhanced by the AVX-512 extension. We evaluate the performance of our approach on a modern Intel Xeon Skylake and assess the different layers of our implementation by sorting/partitioning integers, double floating-point numbers, and key/value pairs of integers. Our results demonstrate that our approach is faster than two libraries of reference: the GNU *C++* sort algorithm by a speedup factor of 4, and the Intel IPP library by a speedup factor of 1.4.**

Keywords*—Quicksort; Bitonic; sort; vectorization; SIMD; AVX-512; Skylake

I. Introduction

Sorting is a fundamental problem in computer science that always had the attention of the research community, because it is widely used to reduce the complexity of some algorithms. Moreover, sorting is a central operation in specific applications such as, but not limited to, database servers [1] and image rendering engines [2]. Therefore, having efficient sorting libraries on new architecture could potentially leverage the performance of a wide range of applications.

The vectorization — that is, the CPU's capability to apply a single instruction on multiple data (SIMD) — improves continuously, one CPU generation after the other. While the difference between a scalar code and its vectorized equivalent was *"only"* of a factor of 4 in the year 2000 (SSE), the difference is now up to a factor of 16 (AVX-512). Therefore, it is indispensable to *vectorize* a code to achieve high-performance on modern CPUs, by using dedicated instructions and registers. The conversion of a scalar code into a vectorized equivalent is straightforward for many classes of algorithms and computational kernels, and it can even be done with auto-vectorization for some of them. However, the opportunity of vectorization is tied to the memory/data access patterns, such that data-processing algorithms (like sorting) usually require an important effort to be transformed. In addition, creating a fully vectorized implementation, without any scalar sections, is only possible and efficient if the instruction set provides the needed operations. Consequently, new instruction sets, such as the AVX-512, allow for the use of approaches that were not feasible previously.

The Intel Xeon Skylake (SKL) processor is the second CPU that supports AVX-512, after the Intel Knight Landing. The SKL supports the AVX-512 instruction set [13]: it supports Intel AVX-512 foundational instructions (AVX-512F), Intel AVX-512 conflict detection instructions (AVX-512CD), Intel AVX-512 byte and word instructions (AVX-512BW), Intel AVX-512 doubleword and quadword instructions (AVX-512DQ), and Intel AVX-512 vector length extensions instructions (AVX-512VL). The AVX-512 not only allows work on SIMD-vectors of double the size, compared to the previous AVX(2) set, it also provides various new operations.

Therefore, in the current paper, we focus on the development of new sorting strategies and their efficient implementation for the Intel Skylake using AVX-512. The contributions of this study are the following:

- proposing a new partitioning algorithm using AVX-512,
- defining a new Bitonic-sort variant for small 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 II gives background information related to vectorization and sorting. We then describe our approach in Section III, introducing our strategy for sorting small arrays, and the vectorized partitioning function, which are combined in our Quicksort variant. Finally, we provide performance details in Section IV and the conclusion in Section V.

II. Background

A. Sorting Algorithms

1) Quicksort (QS) Overview: QS was originally proposed in [3]. It uses a *divide-and-conquer* strategy, by recursively

[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.

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 greater values 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 partitioning pivot, which 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. Fig. 1 shows an example of a QS execution.

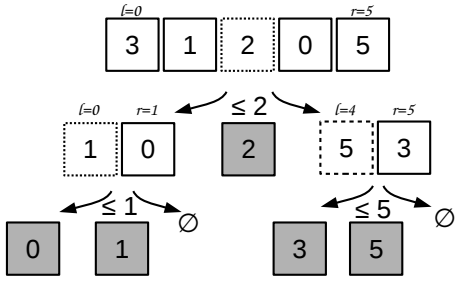


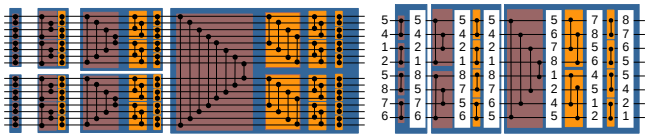
Fig. 1: Quicksort example to sort $[3, 1, 2, 0, 5]$ to $[0, 1, 2, 3, 5]$. The pivot is equal to the value in the middle: the first pivot is 2, then at second recursion level it is 1 and 5.

We provide in Appendix A the scalar QS algorithm. Here, the term *scalar* refers to a single value at the opposite of an SIMD vector. In this implementation, the choice of the pivot is naively made by selecting the value in the middle before partitioning, and this can result in very unbalanced partitions. This is why more advanced heuristics have been proposed in the past, like selecting the median from several values, for example.

2) GNU std::sort Implementation (STL): The worst case complexity of QS makes it no longer suitable to be used as a standard *C++* sort. In fact, a complexity of $O(n \log n)$ in average was required until year 2003 [4], but it is now a worst case limit [5] that a pure QS implementation cannot guarantee. Consequently, the current implementation is a 3-part hybrid sorting algorithm *i.e.* it relies on 3 different algorithms[2]. The algorithm uses an Introsort [6] to a maximum depth of $2 \times \log^2 n$ to obtain small partitions that are then sorted using an insertion sort. Introsort is itself a 2-part hybrid of Quicksort and heap sort.

3) Bitonic Sorting Network: In computer science, a sorting network is an abstract description of how to sort a fixed number of values *i.e.* how the values are compared and exchanged. This can be represented graphically, by having each input value as a horizontal line, and each *compare and exchange* unit as a vertical connection between those lines. There are various examples of sorting networks in the literature, but we concentrate our description on the Bitonic sort from [7]. This network is easy to implement and has an algorithm complexity of $O(n \log(n)^2)$. It has demonstrated good performances on parallel computers [8] and GPUs [9]. Fig. 2a shows a Bitonic sorting network to process 16 values. A sorting network can

[2]See the libstdc++ documentation on the sorting algorithm available at https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html#l05207



(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.

Fig. 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.

be seen as a time line, where input values are transferred from left to right, and exchanged if needed at each vertical bar. We illustrate an execution in Fig. 2b, where we print the intermediate steps while sorting an array of 8 values. The Bitonic sort is not stable because it does not maintain the original order of the values.

If the size of the array to sort is known, it is possible to implement a sorting network by hard-coding the connections between the lines. This can be seen as a direct mapping of the picture. However, when the array size is unknown, the implementation can be made more flexible by using a formula/rule to decide when to compare/exchange values.

B. Vectorization

The term vectorization refers to a CPU's feature to apply a single operation/instruction to a vector of values instead of only a single value [10]. It is common to refer to this concept by 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 single cores, despite the stagnation of the clock frequency. The same strategy is used on new hardware, where the length of the SIMD registers has continued to increase. In the rest of the paper, we use the term *vector* for the data type managed by the CPU in this sense. It has no relation to an expandable vector data structure, such as *std::vector*. The size of the vectors is variable and depends on both the instruction set and the type of vector element, and corresponds to the size of the registers in the chip. Vector extensions to the *x86* instruction set, for example, are SSE [11], AVX [12], and AVX512 [13], which support vectors of size 128, 256 and 512 bits, respectively. This means that an SSE vector is able to store four single precision floating point numbers or two double precision values. Fig. 3 illustrates the difference between a scalar summation and a vector summation for SSE or AVX, respectively. An AVX-512 SIMD-vector is able to store 8 double precision floating-point numbers or 16 integer values, for example. 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 are intended to be replaced with a single assembly instruction by the compiler.

1) AVX-512 Instruction Set: As previous *x86* vectorization extensions, the AVX-512 has instructions to load a contiguous block of values from the main memory and to transform it

Fig. 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.

into a SIMD-vector (load). It is also possible to fill a SIMD-vector with a given value (set), and move back a SIMD-vector into memory (store). A permutation instruction allows to re-order the values inside a SIMD-vector using a second integer array which contains the permutation indexes. This operation was possible in since AVX/AVX2 using *permutevar8x32* (instruction vperm(d,ps)). The instructions *vminpd/vpminsd* return a SIMD-vector where each value correspond to the minimum of the values from the two input vectors at the same position. It is possible to obtain the maximum with instructions *vpmaxsd/vmaxpd*.

In AVX-512, the value returned by a test/comparison (*vpcmpd/vcmppd*) is a mask (integer) and not an 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. Among the mask-based instructions, the *mask move (vmovdqa32/vmovapd)* allows for the selection of values between two vectors, using a mask. Achieving the same result was possible in previous instruction sets using the *blend* instruction since SSE4, and using several operations with previous instruction sets.

The AVX-512 provides operations that do not have an equivalent in previous extensions of the *x86* instruction sets, such as the *store-some* (*vpcompressps/vcompresspd*) and *load-some* (*vmovups/vmovupd*). The *store-some* operation allows to save only a part of a SIMD-vector into memory. Similarly, the *load-some* allows to load less values than the size of a SIMD-vector from the memory. The values are loaded/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.

C. Related Work on Vectorized Sorting Algorithms

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

The sorting technique from [14] 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 was different at that time). This study illustrates the early strategy to adapt sorting algorithms to a given hardware, and also shows the need for low-level optimizations, due to the limited instructions available at that time.

In [15], the authors propose a parallel sorting on top of combosort vectorized with the VMX instruction set of IBM architecture. Unaligned memory access is avoided, and the L2 cache is efficiently managed by using an out-of-core/blocking scheme. The authors show a speedup by a factor of 3 against the GNU *C++* STL.

In [16], the authors use a sorting-network for small-sized arrays, similar to our own 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 opposite. They apply multiple small sorts on sub-parts of the array, and then they finish with a complicated merge scheme using extra memory to globally sort all the sub-parts. A very similar approach was later proposed in [17].

The recent work in [18] targets AVX2. The authors use a Quicksort variant with a vectorized partitioning function, and an insertion sort once the partitions are small enough (as the STL does). The partition method relies on look-up tables, with a mapping between the comparison's result of an 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, but their approach is not always faster than the Intel IPP library. The proposed method is not suitable for AVX-512 because the lookup tables will occupy too much memory. This issue, as well as the use of extra memory, can be solved with the new instructions of the AVX-512. 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.

III. Sorting with AVX-512

A. Bitonic-Based Sort on AVX-512 SIMD-Vectors

In this section, we describe our method to sort small arrays that contain less than 16 times *VEC_SIZE*, where *VEC_SIZE* is the number of values in a SIMD-vector. This function is later used in our final QS implementation to sort small enough partitions.

1) Sorting one SIMD-vector: To sort a single vector, we perform the same operations as the ones shown in Fig. 2a: we compare and exchange values following the indexes from the Bitonic sorting network. However, thanks to the vectorization, we are able to work on the entire vector without having to iterate on the values individually. We know the positions that we have to compare and exchange at the different stages of the algorithm. This is why, in our approach, we rely on static (hard-coded) permutation vectors, as shown in Algorithm 1. In this algorithm, the *compare_and_exchange* function performs all the *compare and exchange* that are applied at the same time in the Bitonic algorithm *i.e.* the operations that are at the same horizontal position in the figure. To have a fully vectorized function, we implement the *compare_and_exchange* in three steps. First, we permute the input vector *v* into *v'* with the given permutation indexes *p*. Second, we obtain two vectors w_{min} and w_{max} that contain the minimum and maximum values between both *v* and *v'*. Finally, we selects the values from w_{min} and w_{max} with a mask-based move, where the mask indicates in which direction the exchanges have to be done. The *C++* source code of a fully vectorized branch-free implementation is given in Appendix B (Code 1).

Algorithm 1: 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 simd_bitonic_sort_1v(vec)
2      compare_and_exchange(vec, [6, 7, 4, 5, 2, 3, 0, 1])
3      compare_and_exchange(vec, [4, 5, 6, 7, 0, 1, 2, 3])
4      compare_and_exchange(vec, [6, 7, 4, 5, 2, 3, 0, 1])
5      compare_and_exchange(vec, [0, 1, 2, 3, 4, 5, 6, 7])
6      compare_and_exchange(vec, [5, 4, 7, 6, 1, 0, 3, 2])
7      compare_and_exchange(vec, [6, 7, 4, 5, 2, 3, 0, 1])
```

2) Sorting more than one SIMD-vectors: The principle of using static permutation vectors to sort a single SIMD-vector can be applied to sort several SIMD-vectors. In addition, we can take advantage of the repetitive pattern of the Bitonic sorting network to re-use existing functions. More precisely, to sort V vectors, we re-use the function to sort $V/2$ vectors and so on. We provide an example to sort two SIMD-vectors in Algorithm 2, where we start by sorting each SIMD-vector individually using the *bitonic_simd_sort_1v* function. Then, we compare and exchange values between both vectors (line 5), and finally applied the same operations on each vector individually (lines 6 to 11). In our sorting implementation, we provide the functions to sort up to 16 SIMD-vectors, which correspond to 256 integer values or 128 double floating-point values.

Algorithm 2: 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 vectors sorted with vec1 lower or equal
           than vec2.
1  function simd_bitonic_sort_2v(vec1, vec2)
2      // Sort each vector using bitonic_simd_sort_1v
3      simd_bitonic_sort_1v(vec1)
4      simd_bitonic_sort_1v(vec2)
5      compare_and_exchange_2v(vec1, vec2, [0, 1, 2, 3, 4, 5, 6, 7])
6      compare_and_exchange(vec1, [3, 2, 1, 0, 7, 6, 5, 4])
7      compare_and_exchange(vec2, [3, 2, 1, 0, 7, 6, 5, 4])
8      compare_and_exchange(vec1, [5, 4, 7, 6, 1, 0, 3, 2])
9      compare_and_exchange(vec2, [5, 4, 7, 6, 1, 0, 3, 2])
10     compare_and_exchange(vec1, [6, 7, 4, 5, 2, 3, 0, 1])
11     compare_and_exchange(vec2, [6, 7, 4, 5, 2, 3, 0, 1])
```

3) Sorting small arrays: Each of our SIMD-Bitonic-sort functions are designed for a specific number of SIMD-vectors. However, we intend to sort arrays that do not have a size multiple of the SIMD-vector's length, because they are obtained from the partitioning stage of the QS. Consequently, when we have to sort a small array, we first load it into SIMD-vectors, and then, we pad the last vector with the greatest possible value. This guarantee that the padding values have no impact on the sorting results by staying at the end of the last vector. The selection of appropriate SIMD-Bitonic-sort function, that matches the size of the array to sort, can be done efficiently with a switch statement. In the following, we refer to this interface as the *simd_bitonic_sort_wrapper* function.

B. Partitioning with AVX-512

Algorithm 3 shows our strategy to develop a vectorized partitioning method. This algorithm is similar to a scalar partitioning function: there are iterators that start from both extremities of the array to keep track of where to load/store the values, and the process stops when some of these iterators meet. In its steady state, the algorithm loads an SIMD-vector using the left or right indexes (at lines 19 and 24), and partitions it using the *partition_vec* function (at line 27). The *partition_vec* function compares the input vector to the pivot vector (at line 47), and stores the values — lower or greater — directly in the array using a store-some instruction (at lines 51 and 55). The store-some is an AVX-512 instruction that we described in Section II-B1. The initialization of our algorithm starts by loading one vector from each array's extremities to ensure that no values will be overwritten during the steady state (lines 12 and 16). This way, our implementation works in-place and only needs three SIMD-vectors. Algorithm 3 also includes, as side comments, possible optimizations in case the array is more likely to be already partitioned (A), or to reduce the data displacement of the values (B). The AVX-512 implementation of this algorithm is given in Appendix B (Code 2). One should note that we use a scalar partition function if there are less than $2 \times$ VEC_SIZE values in the given array (line 3).

C. Quicksort Variant

Our QS is given in Algorithm 4, where we partition the data using the *simd_partition* function from Section III-B, and then sort the small partitions using the *simd_bitonic _sort_wrapper* function from Section III-A. The obtained algorithm is very similar to the scalar QS given in Appendix A.

D. Sorting Key/Value Pairs

The previous sorting methods are designed to sort an array of numbers. However, some applications need to sort key/value pairs. More precisely, the sort is applied on the keys, and the values contain extra information and could be pointers to arbitrary data structures, for example. Storing each key/value pair contiguously in memory is not adequate for vectorization because it requires transforming the data. Therefore, in our approach, we store the keys and the values in two distinct arrays. To extend the SIMD-Bitonic-sort and SIMD-partition functions, we must ensure that the same permutations/moves are applied to the keys and the values. For the partition function, this is trivial. The same mask is used in combination with the store-some instruction for both arrays. For the Bitonic-based sort, we manually apply the permutations that were done on the vector of keys to the vector of values. To do so, we first save the vector of keys *k* before it is permuted by a *compare and exchange*, using the Bitonic permutation vector of indexes *p*, into *k'*. We compare *k* and *k'* to obtain a mask *m* that expresses what moves have been done. Then, we permute our vector of values *v* using *p* into *v'*, and we select the correct values between *v* and *v'* using *m*. Consequently, we perform this operation at the end of the *compare_and_exchange* in all the Bitonic-based sorts.

IV. Performance Study

A. Configuration

We asses our method on an Intel(R) Xeon(R) Platinum 8170 *Skylake* CPU at 2.10GHz, with caches of sizes 32K-Bytes, 1024K-Bytes and 36608K-Bytes, at levels L1, L2 and L3, respectively. The process and allocations are bound with *numactl –physcpubind=0 –localalloc*. We use the Intel

Algorithm 3: SIMD partitioning. VEC_SIZE is the number of values inside a SIMD-vector of type array's elements.

```
   Input: array: an array to partition. length: the size of array. pivot: the
          reference value
   Output: array: the array partitioned. 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      // Set: Fill a vector with all values equal to pivot
8      pivotvec = simd_set_from_one(pivot)
9      // Init iterators and save one vector 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] = partition_vec(array, val, pivotvec, left_w,
             right_w)
28     end
29     // Process left_val and right_val
30     [left_w, right_w] = partition_vec(array, left_val, pivotvec, left_w,
         right_w)
31     [left_w, right_w] = partition_vec(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 partition_vec(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]
```

compiler 17.0.2 (20170213) with aggressive optimization flag *-O3*.

We compare our implementation against two references. The first one is the GNU STL 3.4.21 from which we use the *std::sort* and *std::partition* functions. The second one is the Intel Integrated Performance Primitives (IPP) 2017 which is a library optimized for Intel processors. We use the IPP radix-based sort (function *ippsSortRadixAscend_[type]_I*). This function require additional space, but it is known as one of the fastest existing sorting implementation.

Algorithm 4: 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      // Test if we must partition again or if we can sort
5      if left + SORT_BOUND < right then
6          pivot_idx = select_pivot_pos(array, left, right)
7          swap(array[pivot_idx], array[right])
8          partition_bound = simd_partition(array, left, right,
             array[right])
9          swap(array[partition_bound], array[right])
10         simd_QS_core(array, left, partition_bound-1)
11         simd_QS_core(array, partition_bound+1, right)
12     else
13         simd_bitonic_sort_wrapper(sub_array(array, left),
             right-left+1)
14     end
```

The test file used for the following benchmark is available online[3], 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). The arrays to sort are populated with randomly generated values.

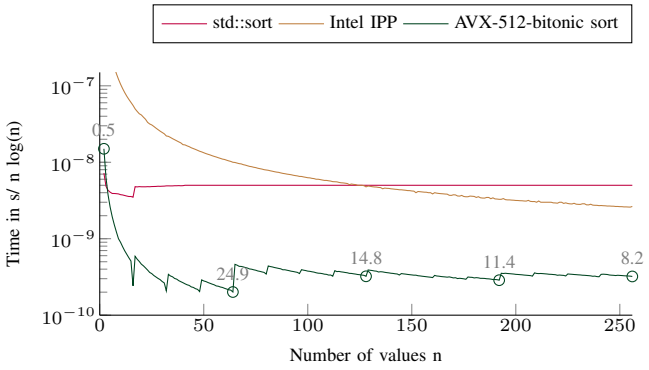
B. Performance to Sort Small Arrays

Fig. 4 shows the execution times to sort arrays of size from 1 to 16 × *VEC_SIZE*, which corresponds to 128 double floating-point values, or 256 integer values. We also test arrays of size not multiple of the SIMD-vector's length. The AVX-512-bitonic always delivers better performance than the Intel IPP for any size, and better performance than the STL when sorting more than 5 values. The speedup is significant, and is around 8 in average. The execution time per item 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, as explained in Section III-A3. For example, in Fig. 4a, the execution time to sort 31 or 32 values is the same, because we sort one SIMD-vector of 32 values in both cases. Our method to sort key/value pairs seems efficient, see Fig. 4c, because the speedup is even better against the STL compared to the sorting of integers.

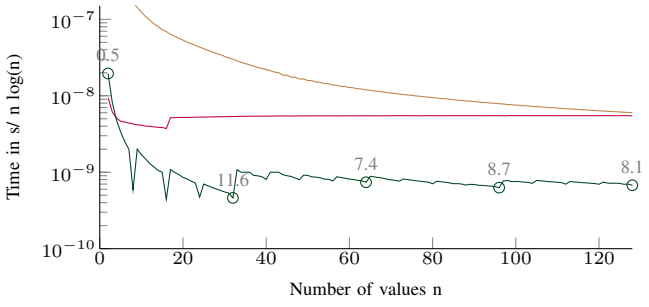
C. Partitioning Performance

Fig. 5 shows the execution times to partition using our AVX-512-partition or the STL's partition function. Our method provides again a speedup of an average factor of 4. For the three configurations, an overhead impacts our implementation and the STL when partitioning arrays larger than 10^7 items. Our AVX-512-partition remains faster, but its speedup decreases from 4 to 3. This phenomena is related to cache effects since 10^7 integers values occupy 40M-Bytes, which is more than the L3 cache size. In addition, we see that this effect starts from 10^5 when partitioning key/value pairs.

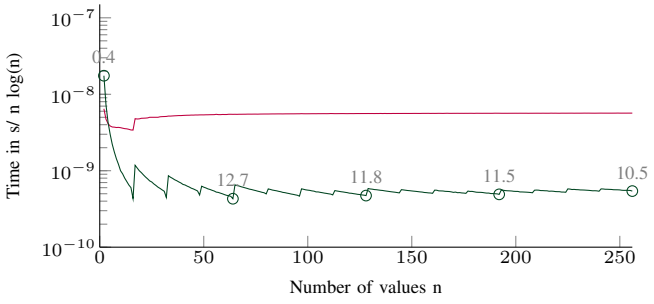
[3]The test file that generates the performance study is available at https://gitlab.mpcdf.mpg.de/bbramas/avx-512-sort (branch paper) under MIT license.



(a) Integer (int).

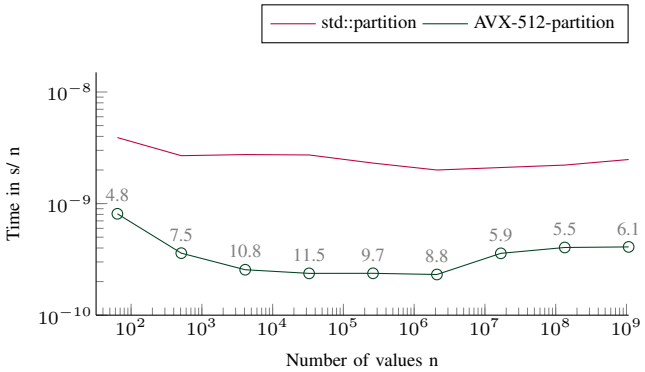


(b) Floating-point (double)

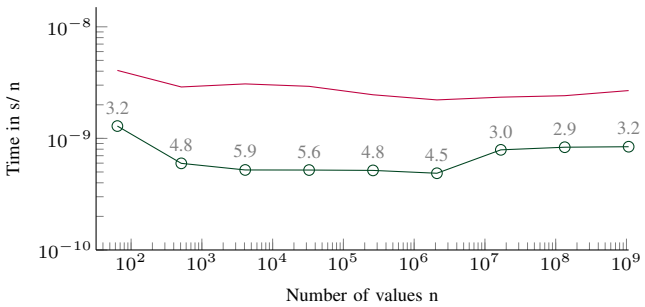


(c) Key/value integer pair (int[2]).

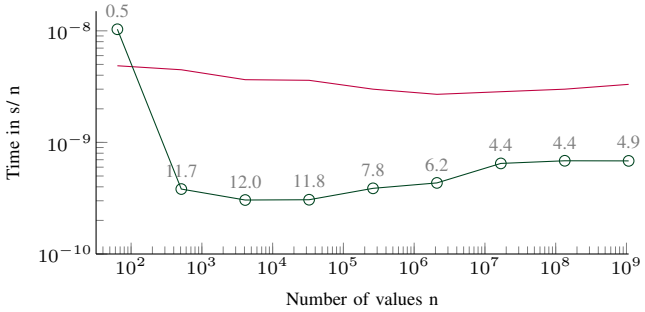
Fig. 4: Execution time divided by *n log(n)* to sort from 1 to 16 × *VEC_SIZE* values. The execution time is obtained from the average of 10^4 sorts for each size. The speedup of the AVX-512-bitonic against the fastest between STL and IPP is shown above the AVX-512-bitonic line.



(a) Integer (int).



(b) Floating-point (double)

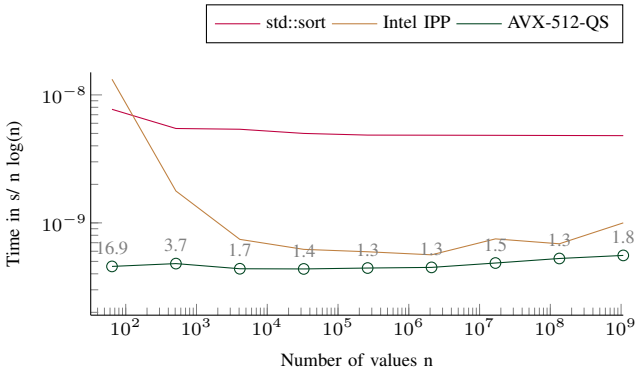


(c) Key/value integer pair (int[2]).

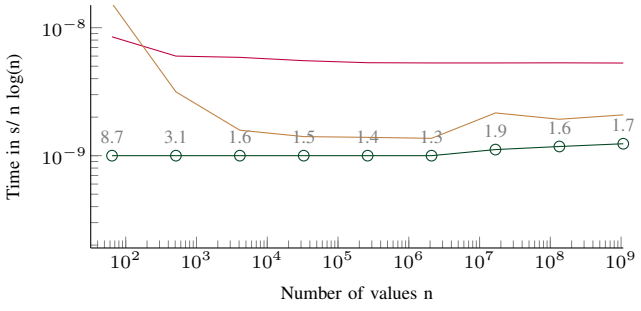
Fig. 5: Execution time divided by n of elements to partition arrays filled with random values with sizes from 2^1 to 2^{30} ($\approx 10^9$).The pivot is selected randomly. the AVX-512-partition line. The execution time is obtained from the average of 20 executions. The speedup of the AVX-512-partition against the STL is shown above.

D. Performance to Sort Large Arrays

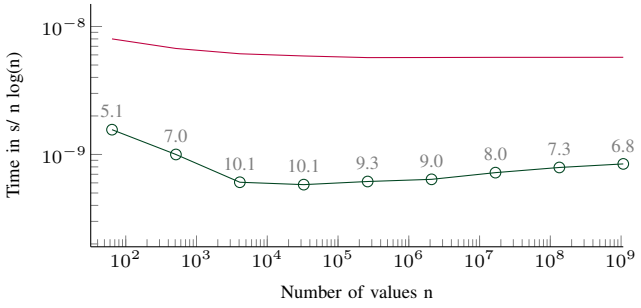
Fig. 6 shows the execution times to sort arrays up to a size of 10^9 items. Our AVX-512-QS is always faster in all configurations. The difference between AVX-512-QS and the STL sort seems stable for any size with a speedup of more than 6 to our benefit. However, while the Intel IPP is not efficient for arrays with less than 10^4 elements, its performance is really close to the AVX-512-QS for large arrays. The same effect found when partitioning appears when sorting arrays larger than 10^7 items. All three sorting functions are impacted, but the IPP seems more slowdown than our method, because it is based on a different access pattern, such that the AVX-512-QS is almost twice as fast as IPP for a size of 10^9 items.



(a) Integer (int)



(b) Floating-point (double)



(c) Key/value integer pair (int[2])

Fig. 6: Execution time divided by *n log(n)* to sort arrays filled with random values with sizes from 2^1 to 2^{30} ($\approx 10^9$). The execution time is obtained from the average of 5 executions. The speedup of the AVX-512-bitonic against the fastest between STL and IPP is shown above the AVX-512-bitonic line.

V. Conclusions

In this paper, we introduced new Bitonic sort and a new partition algorithm that have been designed for the AVX-512 instruction set. These two functions are used in our Quicksort variant which makes it possible to have a fully vectorized implementation (at the exception of partitioning tiny arrays). Our approach shows superior performance on Intel SKL in all configurations against two reference libraries: the GNU *C++* STL, and the Intel IPP. It provides a speedup of 8 to sort small arrays (less than 16 SIMD-vectors), and a speedup of 4 and 1.4 for large arrays, against the *C++* STL and the Intel IPP, respectively. These results should also motivate the community to revisit common problems, because some algorithms may become competitive by being vectorizable, or improved, thanks to AVX-512's novelties. Our source code is publicly available and ready to be used and compared. In the future, we intend to design a parallel implementation of our AVX-512-QS, and we expect the recursive partitioning to be naturally parallelized with a task-based scheme on top of OpenMP.

Appendix

A. Scalar Quicksort Algorithm

Algorithm 5: Quicksort

```
Input: array: an array to sort. length: the size of array.
Output: array: the array sorted.
function QS(array, length)
    QS_core(array, 0, length-1)
function QS_core(array, left, right)
    if left < right then
        // Naive method, select value in the middle
        pivot_idx = ((right-left)/2) + left
        swap(array[pivot_idx], array[right])
        partition_bound = partition(array, left, right, array[right])
        swap(array[partition_bound], array[right])
        QS_core(array, left, partition_bound-1)
        QS_core(array, partition_bound+1, right)
    end
function partition(array, left, right, pivot_value)
    for idx_read ← left to right do
        if array[idx_read] ¡ pivot_value then
            swap(array[idx_read], array[left])
            left += 1
        end
    end
    return left;
```

B. Source Code Extracts

```
inline __m512d AVX_512_bitonic_sort_1v(__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 1: AVX-512 Bitonic sort for one simd-vector of double floating-point values.

```
template <class IndexType>
static inline IndexType AVX_512_partition(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);
}
{
    __mmask8 mask = _mm512_cmp_pd_mask(right_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,right_val);
    left_w += nb_low;

    right_w -= nb_high;
    _mm512_mask_compressstoreu_pd(&array[right_w],~mask,right_val);
}
return left_w;
}
```

Code 2: AVX-512 partitioning of a double floating-point array (AVX-512-partition)

REFERENCES

[1] Graefe, G.: Implementing sorting in database systems. ACM Computing Surveys (CSUR), 38(3):10, 2006.

[2] Bishop, L. Eberly, D. Whitted, T. Finch, M. Shantz,M.: Designing a pc game engine. IEEE Computer Graphics and Applications, 18(1):46–53, 1998.

[3] Hoare,C AR: Quicksort. The Computer Journal, 5(1):10–16, 1962.

[4] ISO/IEC 14882:2003(e): Programming languages - c++, 2003. 25.3.1.1 sort [lib.sort] para. 2.

[5] ISO/IEC 14882:2014(E): Programming Languages - c++, 2014. 25.4.1.1 sort (p. 911).

[6] Musser,D. R.: Introspective sorting and selection algorithms. Softw., Pract. Exper., 27(8):983–993, 1997.

[7] Batcher,K. E.: Sorting networks and their applications. In Proceedings of the April 30–May 2, 1968, spring joint computer conference, pages 307–314. ACM, 1968.

[8] Nassimi, D. Sahni,S.: Bitonic sort on a mesh-connected parallel computer. IEEE Trans. Computers, 28(1):2–7, 1979.

[9] Owens, J. D. Houston, M. Luebke, D. Green, S. Stone, J. E. Phillips, J. C.: Gpu computing. Proceedings of the IEEE, 96(5):879–899, 2008.

[10] Kogge, P. M.: The architecture of pipelined computers. CRC Press, 1981.

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

[12] Intel: Introduction to intel advanced vector extensions. Available on: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[13] Intel: Intel architecture instruction set extensions programming reference. Available on: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[14] Sanders, P. Winkel, S.: Super scalar sample sort. In European Symposium on Algorithms, pages 784–796. Springer, 2004.

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

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

[17] Chhugani, J. Nguyen, A. D. Lee, V. W. Macy, W. Hagog, M. Chen, Y.-K. Baransi, A. Kumar, S. Dubey, P.: Efficient implementation of sorting on multi-core simd cpu architecture. Proceedings of the VLDB Endowment, 1(2):1313–1324, 2008.

[18] Gueron, S. Krasnov, V.: Fast quicksort implementation using avx instructions. The Computer Journal, 59(1):83–90, 2016.