A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE)

Bérenger Bramas

A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE)

Bérenger Bramas
CAMUS, Inria Nancy - Grand Est, Nancy, France
ICPS Team, ICube, Illkirch-Graffenstaden, France
Berenger.Bramas@inria.fr

May 17, 2021
This document is a preprint version. We suggest to have a look at the official repository to see if any new version is available.

1 Abstract

The way developers implement their algorithms and how these implementations behave on modern CPUs are governed by the design and organization of these. The vectorization units (SIMD) are among the few CPUs' parts that can and must be explicitly controlled. In the HPC community, the *x86* CPUs and their vectorization instruction sets were de-facto the standard for decades. Each new release of an instruction set was usually a doubling of the vector length coupled with new operations. Each generation was pushing for adapting and improving previous implementations. The release of the ARM scalable vector extension (SVE) changed things radically for several reasons. First, we expect ARM processors to equip many supercomputers in the next years. Second, SVE's interface is different in several aspects from the *x86* extensions as it provides different instructions, uses a predicate to control most operations, and has a vector size that is only known at execution time. Therefore, using SVE opens new challenges on how to adapt algorithms including the ones that are already well-optimized on *x86*. In this paper, we port a hybrid sort based on the well-known Quicksort and Bitonic-sort algorithms. We use a Bitonic sort to process small partitions/arrays and a vectorized partitioning implementation to divide the partitions. We explain how we use the predicates and how we manage the non-static vector size. We also explain how we efficiently implement the sorting kernels. Our approach only needs an array of $O(logN)$ for the recursive calls in the partitioning phase, both in the sequential and in the parallel case. We test the performance of our approach on a modern ARMv8.2 (A64FX) CPU and assess the different layers of our implementation by sorting/partitioning integers, double floating-point numbers, and key/value pairs of integers. Our

results show that our approach is faster than the GNU *C++* sort algorithm by a speedup factor of 4 on average.

2 Introduction

Sorting is a fundamental problem in computer science and a critical building block for many types of applications such as, but not limited to, database servers [1], image rendering engines [2], mining of time series [3] or JPEG steganography with particle swarm [4]. This is why sorting algorithms have the attention of the research community intending to provide efficient sorting libraries on new architectures that could potentially leverage the performance of a wide range of software. This research of performance is coupled to the changes in CPUs' designs and organization, which includes the vectorization capability.

The performance of CPUs has improved for several decades by increasing the clock frequency. However, this approach has reached a steady-state due to power dissipation and heat effects. To go beyond this limitation, the manufacturers have used parallelization at multiple levels: by embedding multi-cores in a CPU, by allowing pipelining and out-of-order execution at the instruction-level, by putting multiple computational units in each core, and by supporting vectorization. In this context, vectorization consists in the capability of a CPU core of applying a single instruction on multiple data, a concept called SIMD by Flynn's taxonomy [5]. The consequence of this hardware design is that it is mandatory to vectorize a code to achieve high-performance. On the contrary, the throughput can be reduced by at least a factor equivalent to the length of a vector compared to the theoretical peak of the hardware. For instance, the difference between a scalar code and its vectorized equivalent was *"only"* of a factor of 4 in the year 2000 (SSE), but the difference is now up to a factor of 16 (AVX-512) on widespread CPUs.

We can convert many classes of algorithms and computational kernels from a scalar code into a vectorized equivalent without difficulties. Besides, it can be done with auto-vectorization for some of them. However, some algorithms are challenging to adapt because of their memory/data access patterns. Data-processing algorithms (like sorting) are of this kind and require a significant programming effort to be vectorized efficiently. Also, the possibility of creating a fully vectorized implementation, with no scalar sections and with few data transformations, is only possible and efficient if the instruction set extension (IS) provides the needed operations. This is why new ISs together with their new operations make it possible to invent approaches that were not feasible previously, at the cost of reprogramming.

Vectorizing a code can be described as solving a puzzle where the board is the target algorithm and the pieces are the size of the vector and the instructions. However, the paradigm changes with SVE [6, 7, 8] because the size of the vector is unknown at compile time. This can have a significant impact on the transformation from scalar to vectorial. As an example, consider that a developer wants to work on a fixed number of values, which could be linked to

the problem to solve, e.g. a 16×16 matrix-matrix product, or based on other references, e.g. the size of the L1 cache. When the size of the vector is known at development time, a block of data can be mapped to the corresponding number of vectors and working on the vectors can be done with static/known number of operations. With a variable size, it is required to either implement different kernels for each of the possible sizes (like if they were different ISs) or by finding a generic way to vectorize the kernel, which could be a tedious task. We could expect SVE to be less upgraded than *x86* ISs because there will be no need to release a new IS even when new CPU generations will support larger vectors.

In the current paper, we focus on the adaptation of a sorting strategy and its efficient implementation for the ARM CPUs with SVE. Our implementation is generic and works for any size equal to a power of two. The contributions of this study are:

- Describe how we port our AVX-SORT algorithm [9] to SVE;
- Define a new Bitonic-sort variant using SVE and how runtime vector size impact the implementation;
- Implemente an efficient Quicksort variant using OpenMP [10] tasks.

All in all, we show how we can obtain a fast and vectorized in-place sorting implementation. [1]

The rest of the paper is organized as follows: Section 3 gives background information related to vectorization and sorting. We then describe our approach in Section 4, 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 5 and the conclusion in Section 6.

3 Background

3.1 Sorting algorithms

3.1.1 Quicksort (QS) overview

QS [11] is a sorting algorithm that relies on a *divide-and-conquer* strategy. It recursively partitions the input array, until it ends with partitions of one value. The partitioning algorithm consists in moving the 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 choice of the pivot influences the complexity and 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 made it a very popular sorting algorithm. Figure 1 shows an example of a QS execution.

[1] The functions described in the current study are available at https://gitlab.inria.fr/bramas/sve-sort. This repository includes a clean header-only library and a test file that generates the performance study of the current manuscript. The code is under MIT license.

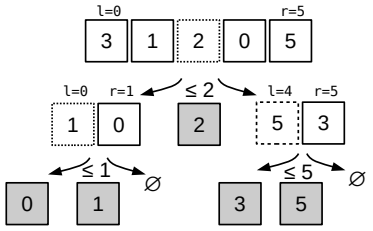


Figure 1: Quicksort example to sort $[3, 1, 2, 0, 5]$ to $[0, 1, 2, 3, 5]$. The pivot is equal to the value in the middle: the first pivot is 2, then at second recursion level it is 1 and 5. l is the left index, and r the right index.

To parallelize the QS and other *divide-and-conquer* approaches, it is common to create a task for each recursive call followed by a wait statement. For instance, a thread partitions the array in two, and then creates two tasks (one for each of the partition). To ensure coherency, the thread waits for the completion of the tasks before continuing. We refer to this parallel strategy as the *QS-par*.

3.1.2 GNU std::sort implementation (STL)

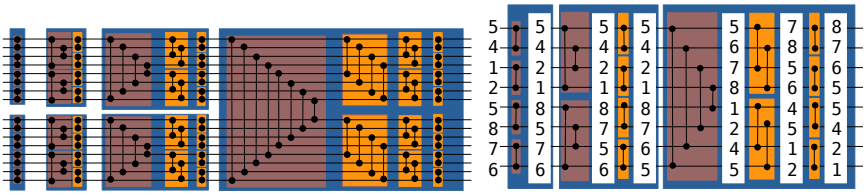
QS is inadequate to a standard *C++* sort because of its worst-case complexity. A complexity of $O(n \log n)$ in average was required until year 2003 [12], but it is now a worst case limit [13] that a pure QS implementation cannot guarantee. The current STL implementation is a 3-part hybrid sorting algorithm *i.e.* it relies on 3 different algorithms [2]. The algorithm uses an Introsort [14] to a maximum depth of $2 \times \log^2 n$ to get small partitions that are then sorted using an insertion sort. Introsort is itself a 2-part hybrid of Quicksort and heap sort.

3.1.3 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 [15]. 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 [16] and GPUs [17]. Figure 2(a) shows a Bitonic sorting network to process 16 values. A sorting network can be seen as a timeline, where input values are transferred from left to right, and exchanged if needed at each vertical bar. We illustrate an execution in Figure 2(b), where we print the intermediate steps while sorting an array of 8 values. We use the terms *symmetric* and *stair* exchanges to refer to the red

[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

and orange stages, respectively. A *symmetric* stage is always followed by *stair* stages from half size to size two. The Bitonic sort does not maintain the original order of the values and thus is not stable.



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 and we refer to it as the *symmetric* stage. Whereas in orange boxes, the exchanges are done with a linear progression and we refer to it as the *stair* stage.

If we know the size of the array to sort, it is possible to implement a sorting network by hard-coding the connections between the lines. We can see this as a direct mapping of the picture. When the array size is unknown, the implementation can be made more flexible by using a formula to decide when to compare and exchange the values [18].

3.2 Vectorization

The term vectorization refers to a CPU feature of applying a single operation/instruction to a vector of values, which is the opposite of scalar when an instruction applies to only a single value [19]. Despite the stagnation of the clock frequency since the mid-2000s, it has been possible to increase the peak performance of single cores by adding SIMD instructions/registers to CPUs, among others. The length of the SIMD registers has continuously increased, allowing the performance of the chips to increase accordingly. 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's elements, and corresponds to the size of the registers in the chip.

The SIMD instructions can be called in the assembly language or using *intrinsic* functions, which are small functions that are intended to be replaced with a single assembly instruction by the compiler. There is usually a one-to-one mapping between intrinsics and assembly instructions, but this is not always true as some intrinsics are converted into several instructions. Moreover, the compiler is free to use different instructions as long as they give the same

results.

3.2.1 x86 instruction set extensions

Vector extensions to the *x86* instruction set have been massively used in HPC. For example, the common extensions are SSE [20], AVX [21], and AVX-512 [22], which support vectors of size 128, 256 and 512 bits respectively. This means that an SSE vector can store four single-precision floating-point numbers or two double-precision values. Figure 3 illustrates the difference between a scalar summation and a vector summation for SSE or AVX, respectively.

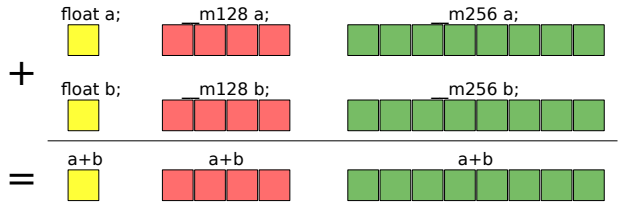


Figure 3: Summation example of single precision floating-point values using : (■) scalar standard *C++* code, (■) SSE SIMD-vector of 4 values , (■) AVX SIMD-vector of 8 values.

3.2.2 SVE instruction set

The SVE is a feature for ARMv8 processors. The size of the vector is not fixed at compile time (the specification limits the size to 2048 bits and ensures that it is a multiple of 128 bits) such that a binary that includes SVE instructions can be executed on ARMv8 that support SVE no matter the size of their registers. SVE provides most classic operations that also exist in *x86* vectorization extensions, such as loading a contiguous block of values from the main memory and transforming it into a SIMD-vector (load), filling a SIMD-vector with a value (set), move back a SIMD-vector into memory (store) and basic arithmetic operations. SVE also provides advanced operations like gather, scatter, indexed accesses, permutations, comparisons, and conversions. It is also possible to get the maximum or the minimum of a vector or element-wise between two vectors.

Another significant difference with other ISs, is the use of predicate vectors i.e. the use of Boolean vectors (*svbool_t*) that allow controlling more finely the instructions by selecting the affected elements, for example. Also, while in AVX-512 the value returned by a test/comparison (*vpcmpd/vcmppd*) is a mask (integer), in SVE, the result is *svbool_t*.

A minor difference, but which impacts our implementation, is that SVE does not support a *store-some* as it exists in AVX-512 (*vpcompressps/vcompresspd*), where some values of a vector can be stored contiguously in memory. With SVE it is needed to first compact the values of a vector to put the values to be saved at the beginning of the vector, and then perform a store, or to use a scatter.

However, both approaches need extra Boolean or indices vectors and additional instructions.

3.3 Related work on vectorized sorting algorithms

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

The sorting technique from Sanders et al. [23] 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). 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.

In [24], 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 a different study [25], 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 sort globally all the sub-parts. A very similar approach was later proposed by Chhugani et al. [26].

A more recent work targets AVX2 [27]. 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 show 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, and 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. It is the only part of their algorithm that is vectorized.

In our previous work [9], we have proposed the first hybrid QS/Bitonic algorithm implemented with AVX-512. We have described how we can vectorize the partitioning algorithm and create a branch-free/vectorized Bitonic sorting kernel. To do so, we put the values of the input array into SIMD vectors. Then, we sort each vector individually and finally we exchange values between vectors either during the *symmetric* or *stair* stage. Our method was 8 times faster to sort small arrays and 1.7 times faster to sort large arrays compared to the Intel IPP library. However, our method was sequential and could not simply be

converted to SVE when we consider that the vector size is unknown at compile time. In this study, we refer to this approach as the *AVX-512-QS*.

Hou et al. [28] designed a framework for the automatic vectorization of parallel sort on *x86*-based processors. Using a DSL, their tool generates a SIMD sorting network based on a formula. Their approach shows a significant speedup against STL, and especially they show a speedup of 6.7 in parallel against the sort from Intel TBB on Intel Knights Corner MIC. The method is of great interest as it avoids to program by hand the core of the sorting kernel. Any modification, such as the use of a new IS, requires upgrading the framework. To the best of our knowledge, they do not support SVE yet.

Yin et al. [29] described an efficient parallel sort on AVX-512-based multi-core and many-core architectures. Their approach achieves to sort 1.1 billion floats per second on an Intel KNL (AVX-512). Their parallel algorithm is similar to the one we use in the current study because they first sort sub-parts of the input array and then merge them by pairs until there is only one result. However, their parallel merging is out-of-place and requires doubling the needed memory, which is not the case for us. Besides, their Bitonic sorting kernel differs from ours, because we follow the Bitonic algorithm without the need for matrix transposition inside the registers.

Watkins et al. [30] provide an alternative approach to sort based on the merging of multiple vectors. Their method is 2 times faster than the Intel IPP library and 5 times faster than the C-lib *qsort*. They can sort 500 million keys per second on an Intel KNL (AVX-512) but they also need to have an external array when merging, which we avoid in our approach.

4 Sorting with SVE

4.1 Overview

Our SVE-QS is similar to the AVX-512-QS and is composed of two key steps. First, we partition the data recursively using the *sve_partition* function described in Section 4.3, as in the classical QS. Second, once the partitions are smaller than a given threshold, we sort them with the *sve_bitonic_sort_wrapper* function from Section 4.2. To sort in parallel, we rely on the classical parallelization scheme for the divide-and-conquer algorithm but propose several optimizations. This allows an easy parallelization method, which can be fully implemented using OpenMP.

4.2 Bitonic-based sort on SVE 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 vector. This function is later used in our final QS implementation to sort small enough partitions.

4.2.1 Sorting one vector

To sort a single vector, we perform the same operations as the ones shown in Figure 2(a): we compare and exchange values following the indexes from the Bitonic sorting network. Thanks to the vectorization, we can work on the entire vector without having to iterate on the values individually. However, we cannot hard-code the indices of the elements that should be compared and exchanged, because we do not know the size of the vector. Therefore, we use a loop-based scheme where we efficiently generate permutation and Boolean vectors to perform the correct comparisons. We use the same pattern for both the *symmetric* and the *stair* stages.

In the *symmetric* stage, the values are first compared by contiguous pairs, e.g. each value at an even index i is compared with the value at $i + 1$ and each value at en odd index j is compared with the value at $j - 1$. Additionally, we see in Figure 2(a) that the width of comparison doubles at each iteration and that the comparisons are from the sides to the center. In our approach, we use three vectors. First, a Boolean vector that shows the direction of the comparisons, e.i. for each index it tells if it has to be compared with a value at a greater index (and will take the minimum of both) or with a value at a lower index (and will take the maximum of both). Second, we need a shift coefficient vector which gives the step of the comparisons, i.e. it tells for each index the relative position of the index to be compared with. Finally, we need an index vector that contains increasing values from 0 to N-1 (for any index i, $vec[i] = i$) and that we use to sum with the shift coefficient vector to get a permutation vector. The permutation vector tells for any index which other index it should be compared against.

We give the pseudo-code of our vectorized implementation in Algorithm 1 where the corresponding SVE instructions and possible vector values are written in comments. In the beginning, the Boolean vector must contain repeated *false* and *true* values because the values are compared by contiguous pairs. To build it, we use the *svzip1* instruction which interleaves elements from low halves of two inputs, and pass one vector of *true* and a vector of *false* as parameters (line 7). Then, at each iteration, the number of *false* should double and be followed by the same number of *true*. To do so, we use again the *svzip1* instruction but we pass the Boolean vector as parameters (line 20). The vector of increasing indexes is built with a single SVE instruction (line 5). The shift coefficients vector is built by interleaving 1 and -1 (line 9). The permutation index is generated by summing the two vectors (line 12) and uses to permute the input (line 14). So, at each iteration, we use the updated Boolean vector to decide if we add or subtract two times the iteration index (line 22). Also, this algorithm is never used as presented here because each of its iterations must be followed by a *stair* stage.

We use the same principle in the *stair* stage with one vector for the Boolean that shows the direction of the exchange and another to store the relative index for comparison. We sum this last vector with the index vector to get the permutation indices. If we study again to Figure 2(a), we observe that the al-

Algorithm 1: SVE Bitonic sort for one vector, symmetric stage.

```
   Input: vec: a SVE vector to sort.
   Output: vec: the vector sorted.
 1 function sve_bitonic_sort_1v_symmetric(vec)
 2     // Number of values in a vector
 3     N = get_hardware_size()
 4     // svindex - [O, 1, ...., N-1]
 5     vecIndexes = (i ∈ [0, N-1] → i)
 6     // svzip1 - [F, T, F, T, ..., F, T]
 7     falseTrueVecOut = (i ∈ [0, N-1] → i is odd ? False : True)
 8     // svneg/svdup - [1, -1, 1, -1, ..., 1, -1]
 9     vecIndexesPermOut = (i ∈ [0, N-1] → falseTrueVecOut[i] ? -1 : 1)
10     for stepOut from 1 to N-1, doubling stepOut at each step do
11         // svadd - [1, 0, 3, 2, ..., N-1, N-2]
12         premuteIndexes = (i ∈ [0, N-1] → vecIndexes[i] + vecIndexesPermOut[i])
13         // svtbl - [vec[1], vec[0], vec[3], vec[2], ..., vec[N-1], vec[N-2]]
14         vecPermuted = (i ∈ [0, N-1] → vec[premuteIndexes[i]])
15         // svsel/svmin/svmax - [..., Min(vec[i], vec[i+1]), Max(vec[i], vec[i+1]), ...]
16         vec = (i ∈ [0, N-1] → falseTrueVecOut[i] ?
17                                 Max(vec[i], vecPermuted[i]):
18                                 Min(vec[i], vecPermuted[i]))
19         // svzip1 - [F, F, T, T, F, F, T, T, ...]
20         falseTrueVecOut = (i ∈ [0, N-1] → falseTrueVecOut[i/2])
21         // svsel/svadd/svsub - [3, 2, 1, 0, 3, 2, 1, 0, ...]
22         vecIndexesPermOut = (i ∈ [0, N-1] → falseTrueVecOut[i] ?
23                                             vecIndexesPermOut[i]-stepOut*2 :
24                                             vecIndexesPermOut[i]+stepOut*2)
25     end
26     return vec
```

gorithm starts by working on parts of half the size of the previous *symmetric* stage. Then, at each iteration, the parts are subdivided until they contain two elements. Besides, the width of the exchange is the same for all elements in an iteration and is then divided by two for the next iteration.

We provide a pseudo-code of our vectorized algorithm in Algorithm 2. To manage the Boolean vector: we use the *svuzp2* instruction that select odd elements from two inputs and concatenate them. In our case, we passe a vector that contains a repeated pattern composed of *false* x times, followed by *true* x times (x a power of two) to *svuzp2* to get a vector with repetitions of size $x/2$. Therefore, we pass the vector of Boolean generated during the *symmetric* stage to *svuzp2* to initialize the new Boolean vector (line 7). We divide the exchange step by two for all elements (line 23). The permutation (line 15) and exchange (line 17) are similar to what is performed in the *symmetric* stage.

The complete function to sort a vector is a mix of the *symmetric* (*sve_bitonic_sort_1v_symmetric*) and *stair* (*sve_bitonic_sort_1v_stairs*) functions; each iteration of the *symmetric* stage is followed by the inner loop of the *stair* stage. The corresponding *C++* source code of a fully vectorized implementation is given in Appendix A.1.

4.2.2 Sorting more than one vectors

To sort more than one vector, we profit that the same patterns are repeated at different scales; to sort V vectors, we re-use the function that sorts $V/2$ vectors and so on. We provide an example to sort two vectors in Algorithm 3, where we

Algorithm 2: SVE Bitonic sort for one vector, *stair* stage. The gray lines are copied from the *symmetric* stage (Algorithm 1)

```
   Input: vec: a SVE vector to sort.
   Output: vec: the vector sorted.
1  function sve_bitonic_sort_1v_stairs(vec)
2      N = get_hardware_size()
3      vecIndexes = (i ∈ [0, N-1] → i)
4      falseTrueVecOut = (i ∈ [0, N-1] → i is odd ? False : True)
5      for stepOut from 1 to N-1, doubling stepOut at each step do
6          // svuzp2
7          falseTrueVecIn = (i ∈ [0, N-1] → falseTrueVecOut[(i*2+1)%N])
8          // svdup - [stepOut/2, stepOut/2, ...]
9          vecIncrement = (i ∈ [0, N-1] → stepOut/2)
10         for stepIn from stepOut/2 to 1, dividing stepIn by 2 at each step do
11             // svadd/svneg - [stepOut/4, stepOut/4,..., -stepOut/4, -stepOut/4]
12             premuteIndexes = (i ∈ [0, N-1] → vecIndexes[i] +
13                                 (falseTrueVecIn[i] ? -vecIncrement[i] : vecIncrement[i]))
14             // svtbl
15             vecPermuted = (i ∈ [0, N-1] → vec[premuteIndexes[i]])
16             // svsel/svmin/svmax
17             vec = (i ∈ [0, N-1] → falseTrueVecIn[i] ?
18                                     Max(vec[i], vecPermuted[i]):
19                                     Min(vec[i], vecPermuted[i]))
20             // svuzp2
21             falseTrueVecIn = (i ∈ [0, N-1] → falseTrueVecIn[(i*2+1)%N])
22             // svdiv
23             vecIncrement = (i ∈ [0, N-1] → vecIncrement[i] / 2);
24         end
25         falseTrueVecOut = (i ∈ [0, N-1] → falseTrueVecOut[i/2])
26     end
27     return vec
```

start by sorting each vector individually using the *sve_bitonic_sort_1v* function. Then, we compare and exchange values between both vectors (line 9), and we finish by applying the same *stair* stage on each vector individually. Our real implementation uses an optimization that consists in a full inlining followed by a merge of the same operations done on different data. For instance, instead of two consecutive calls to *sve_bitonic_sort_1v* (lines 7 and 8), we inline the functions. But since they are similar but on different data, we merge them into one that works on both vectors at the same time. In our sorting implementation, we provide the functions to sort up to 16 vectors.

4.2.3 Sorting small arrays

Each of our SVE-Bitonic functions is designed for a specific number of vectors. However, the partitions obtained from our QS algorithm do not have a size multiple of the vector's length. Therefore, when we sort a small array, we first load it into vectors, and then, we pad the last vector with the greatest possible value. These last values will have no impact on the sorting results because they will stay at the end of the last vector. We select the appropriate SIMD-Bitonic-sort function that matches the size of the array to sort with a switch statement. In the following, we refer to this interface as the *sve_bitonic_sort_wrapper* function.

Algorithm 3: SIMD bitonic sort for two vectors of double floating-point values.

```
   Input: vec1 and vec2: two double floating-point SVE vectors to sort.
   Output: vec1 and vec2: the two vectors sorted with vec1 lower or equal than vec2.
1  function sve_bitonic_exchange_rev(vec1, vec2)
2      vec1_copy = (i ∈ [0, N-1] → vec1[N-1-i])
3      vec1 = (i ∈ [0, N-1] → Min(vec1[i], vec2[i])
4      vec2 = (i ∈ [0, N-1] → Max(vec1_copy[i], vec2[i])
5      return {vec1, vec2}
6  function sve_bitonic_sort_2v(vec1, vec2)
7      vec1 = sve_bitonic_sort_1v(vec1)
8      vec2 = sve_bitonic_sort_1v(vec2)
9      [vec1, vec2] = sve_bitonic_exchange_rev(vec1, vec2)
10     vec1 = sve_bitonic_sort_1v_stairs(vec1)
11     vec2 = sve_bitonic_sort_1v_stairs(vec2)
```

4.2.4 Optimization by comparing vectors' min/max values or whether vectors are already sorted

There are two main points where we can apply optimization in our implementation. The first one is to avoid exchanging values between vectors if their contents are already in the correct order, i.e. no values will be exchanged between the vectors because their values respect the ordering objective. For instance, in Algorithm 3, we can compare if the greatest value in vector *vec2* (SVE instruction *svmaxv*) is lower than or equal to the lowest value in vector *vec1* (SVE instruction *svminv*). If this is the case the function can simply sort each vector individually. The same mechanism can be applied to any number of vectors, and it can be used at function entry or inside the loops to break when it is known that no more values will be exchanged. The second optimization can be applied when we want to sort a single vector by checking if it is already sorted. Similarly to the first optimization, this check can be done at function entry or in the loops, such as at lines 2 and 10, in Algorithm 2. We propose two implementations to test if a vector is sorted and provide the details in Appendix A.2.

4.3 Partitioning with SVE

Our partitioning strategy is based on the AVX-512-partition. In this algorithm, we start by saving the extremities of the input array into two vectors that remain unchanged until the end of the algorithm. By doing so, we free the extremity of the array that can be overwritten. Then, in the core part of the algorithm, we load a vector and compare it to the pivot. The values lower than the pivot are stored on the left side of the array and the values greater than the pivot are stored on the right side while moving the corresponding cursor indexes. Finally, when there is no more value to load, the two vectors that were loaded at the beginning are compared to the pivot and stored in the array accordingly.

When we implement this algorithm using SVE we obtain a Boolean vector b when we compare a vector to partition with the pivot. We use b to compact the vector and move the values lower or equal than the pivot on the left, and then we generate a secondary Boolean vector to store only as a sub-part of the

vector. We manage the values greater than the pivot similarly by using the negate of b.

4.4 Sorting key/value pairs

The sorting methods we have described are designed to sort arrays 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 such as pointers to arbitrary data structures, for example. We extend our SVE-Bitonic and SVE-Partition functions by making sure that the same permutations/moves apply to the keys and the values. In the sort kernels, we replace the minimum and maximum statements with a comparison operator that gives us a Boolean vector. We use this vector to transform both the vector of keys and the vector of values. For the partitioning kernel, we already use a comparison operator, therefore, we add extra code to apply the same transformations to the vector of values and the vector of keys.

In terms of high-level data structure, we support two approaches. In the first one, we store the keys and the values in two distinct arrays, which allow us to use contiguous load/store. In the second one, the key/value is stored by pair contiguously in a single array, such that loading/storing requires non-contiguous memory accesses.

4.5 Parallel sorting

Our parallel implementation is based on the *QS-par* that we extend with several optimizations. In the *QS-par* parallelization strategy, it is possible to avoid having too many tasks or tasks on too small partitions by stoping creating tasks after a given recursive level. This approach allows to fix the number of tasks at the beginning but could end in an unbalanced configuration (if the tasks have different workload) that is difficult to resolve on the fly. Therefore, in our implementation, we create a task for every partition larger than the L1 cache. However, we do not rely on the OpenMP task statement because it is impossible to control the data locality. Instead, we use one task list per thread. Each thread uses its list as a stack to store the intervals of the recursive calls and also as a task list where each interval can be processed in a task. In a steady-state, each thread accesses only its list: after each partitioning, a thread puts the interval of the first sub-partition in the list and continues with the second sub-partition. When the partition is smaller than the L1 cache, the thread executes the sequential SVE-QS. We use a work-stealing strategy when a thread has an empty list such that the thread will try to pick a task in others' lists. The order of access to others' lists is done such that a thread accesses the lists from threads of closer ids to far ids, e.g. a thread of id i will look at $i+1$, $i-1$, $i+2$, and so on. We refer to this optimized version as the *SVE-QS-par*.

5 Performance study

5.1 Configuration

We assess our method on an ARMv8.2 *A64FX - Fujitsu* with 48 cores at 1.8GHz and 512-bit SVE, i.e. a vector can contain 16 integers and 8 double floating-point values. The node has 32 GB HBM2 memory arranged in 4 core memory groups (CMGs) with 12 cores and 8GB each, 64KB private L1 cache, 8MB shared L2 cache per CMG. For the sequential executions, we pinned the process with *taskset -c 0*, and for the parallel executions, we use *OMP_PROC_BIND=TRUE*. We use the ARM compiler 20.3 (based on LLVM 9.0.1) with the aggressive optimization flag *-O3*. We compare our sequential implementations against the GNU STL 20200312 from which we use the *std::sort* and *std::partition* functions. We also compare against an implementation that we have obtained by performing a translation of our original AVX-512 into SVE. This implementation works only for 512-bit SVE. We compare our parallel implementation against the Boost [3] 1.73.0 from which we use the *block_indirect_sort* function. The test file used for the following benchmark is available online (`https://gitlab.inria.fr/bramas/arm-sve-sort`) and includes the different sorts presented in this study plus some additional strategies and tests.[4] Our QS uses a 5-values median pivot selection (whereas the STL sort function uses a 3-values median). The arrays to sort are populated with randomly generated values. Our implementation does not include the potential optimizations described in Section 4.2.4 that can be applied when there is a chance that parts or totality of the input array are already sorted.[5]

5.2 Performance to sort small arrays

Figure 4 shows the execution times to sort arrays of size from 1 to 16 $\times$ *VEC_SIZE* (which corresponds to 128 double floating-point values, or 256 integer values) by step 1, such that we also test arrays of size not multiple of the SIMD-vector's length. The SVE-Bitonic always delivers better performance than the STL when sorting more than 20 values. The speedup is significant and increases with the number of values to reach 5 for 256 integer values. 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 4.2.3. For example, the execution time to sort 31 or 32 integers is the same, because we sort two SIMD-vector of 16 values in both cases. Our method to sort key/value pairs seems efficient, and the speedup reaches 3.6. To sort key/value pairs we obtain similar performance if we sort pair of integers stored contiguously or two arrays of integers, one for the keys and one for the values. Comparing our two SVE implementations, SVE-Bitonic appears more

[3] https://www.boost.org/

[4] It can be executed on any CPU using the Farm-SVE library (`https://gitlab.inria.fr/bramas/farm-sve`).

[5] This implementation is partially implemented in the branch *optim* of the code repository.

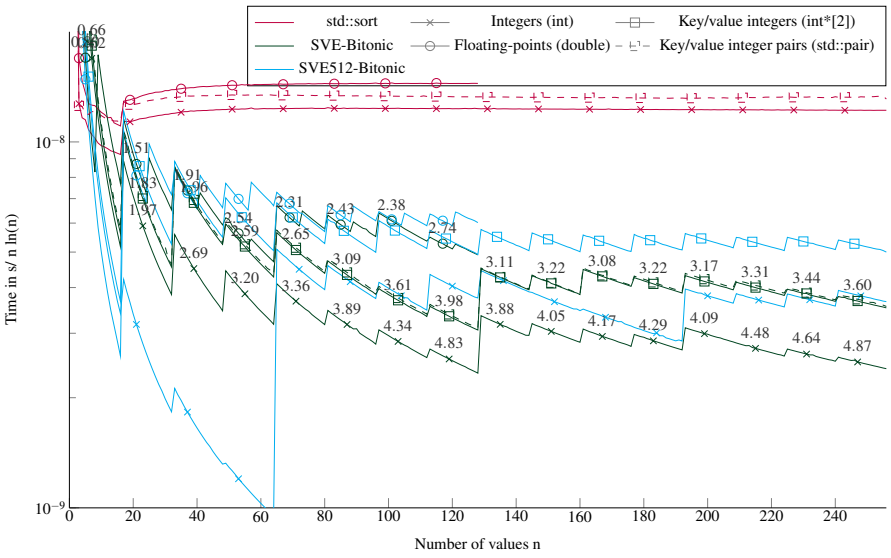


Figure 4: Execution time divided by $n\ ln(n)$ to sort from 1 to 16 $\times$ *VEC_SIZE* values. The execution time is obtained from the average of $2 \cdot 10^3$ sorts with different values for each size. The speedup of the SVE-Bitonic against the STL is shown above the SVE-Bitonic lines. Key/value integers as a *std::pair* are plot with dashed lines and as two distinct integer arrays (int*[2]) are plot with dense lines.

efficient than SVE512-bitonic, except for very small number of values. This means that considering a static vector size of 512 bits, with compare-exchange indices hard coded and no loops/branches, does not provide any benefit, and is even slower for more than 70 values. This means that, for our kernels, the CPU manages more easily loops with branche (SVE-Bitonic) than a large amount of instructions without branches (SVE512-bitonic). Sorting double floating-points values or pairs of integers takes similar duration up to 64 values, then with more values it is faster to sort pairs of integers.

5.3 Partitioning performance

Figure 5 shows the execution times to partition using our SVE-Partition or the STL's partition function. Our method provides again a speedup of an average factor of 4 for integers and key/values (with two arrays), and 3 for floating-point values. We see no difference if the data fit in the caches L1/L2 or not, neither in terms of performance nor in the difference between the STL and our implementation. However, there is a significant difference between partitioning two arrays of integers (one for the key and the other for the values) or one array of pairs of integers. The only difference between both implementations is that we work with distinct svint32_t vectors in the first one, and with svint32x2_t vector

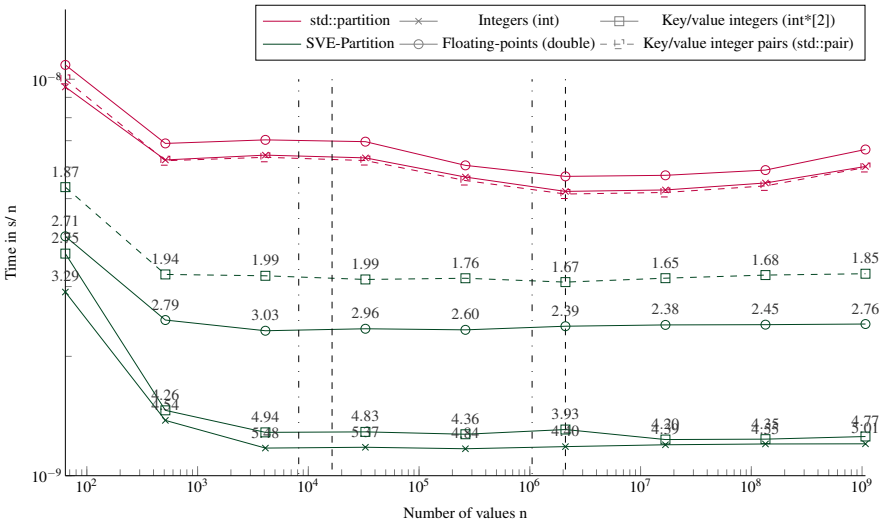


Figure 5: Execution time divided by n of elements to partition arrays filled with random values with sizes from 64 to $\approx 10^9$ elements.The pivot is selected randomly. The execution time is obtained from the average of 20 executions with different values. The speedup of the SVE-partition against the STL is shown above the lines. The vertical lines represent the caches relatively to the processed data type ($-$ for the integers and $\cdot - \cdot$ for floating-points and the key/value integers). Key/value integers as a *std::pair* are plot with dashed lines and as two distinct integer arrays (int*[2]) are plot with dense lines.

pairs in the second. But the difference is mainly in the memory accesses during the loads/stores. The partitioning of one array or two arrays of integers appears equivalent, and this can be unexpected because we need more instructions when managing the later. Indeed, we have to apply the same transformations to the keys and the values, and we have twice memory accesses.

5.4 Performance to sort large arrays

Figure 6 shows the execution times to sort arrays up to a size of $\approx 10^9$ items. Our SVE-QS is always faster in all configurations. The difference between SVE-QS and the STL sort is stable for size greater than 10^3 values with a speedup of more than 4 to our benefit to sort integers. There is an effect when sorting 64 values (the left-wise points) as the execution time is not the same as the one observed when sorting less than 16 vectors (Figure 4). The only difference is that here we call the main SVE-QS functions, which call the SVE-Bitonic functions after just one test on the size, whereas in the previous results we call the SVE-Bitonic functions directly. We observe that when sorting key/value pairs there is again a benefit when using two distinct arrays of scalars compared with a single array of pairs. From the previous results, it is clear that this difference comes from the

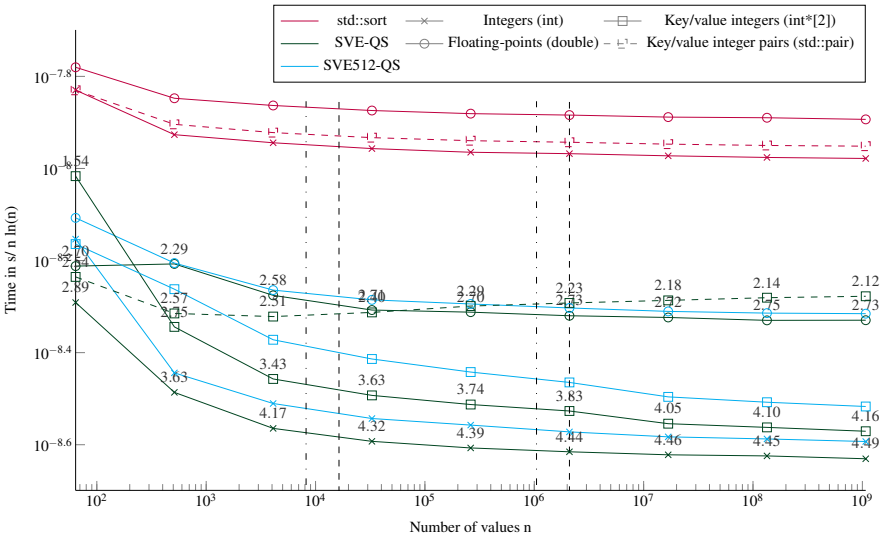


Figure 6: Execution time divided by *n ln(n)* to sort arrays filled with random values with sizes from 64 to $\approx 10^9$ elements. The execution time is obtained from the average of 5 executions with different values. The speedup of the SVE-QS against the STL is shown above the SVE-QS lines. The vertical lines represent the caches relatively to the processed data type (− for the integers and $\cdot - \cdot$ for floating-points and the integer pairs). Key/value integers as a *std::pair* are plot with dashed lines and as two distinct integer arrays (int*[2]) are plot with dense lines.

partitioning for which the difference also exists (Figure 5), whereas the difference is negligible in the sorting of arrays smaller than 16 vectors (Figure 4). However, as the size of the array increases, this difference vanishes, and it becomes even faster to sort Floating-point values than keys/values.

5.5 Performance of the parallel version

Figure 7 shows the performance for a different number of threads of a parallel sort implementation from the boost library (block_indirect_sort) against our task-based implementation (SVE-QS-par). The 1 thread executions show that our SVE-QS-par is faster for both data types. Then, as the number of threads increases, the SVE-QS-par becomes faster but reaches a limit at 16 threads, and using 32 or 48 threads does not provide a significant benefit. The three curves for 16, 32, and 48 threads, overlap for the Integers and join at size 10^8 for Floating-point values. The block_indirect_sort implementation becomes competitive at 10^8 and is faster than our approach for Floating-point values. This illustrates the limit of the divide-and-conquer parallelization strategy to process large arrays, whereas the block_indirect_sort also uses a merge kernel

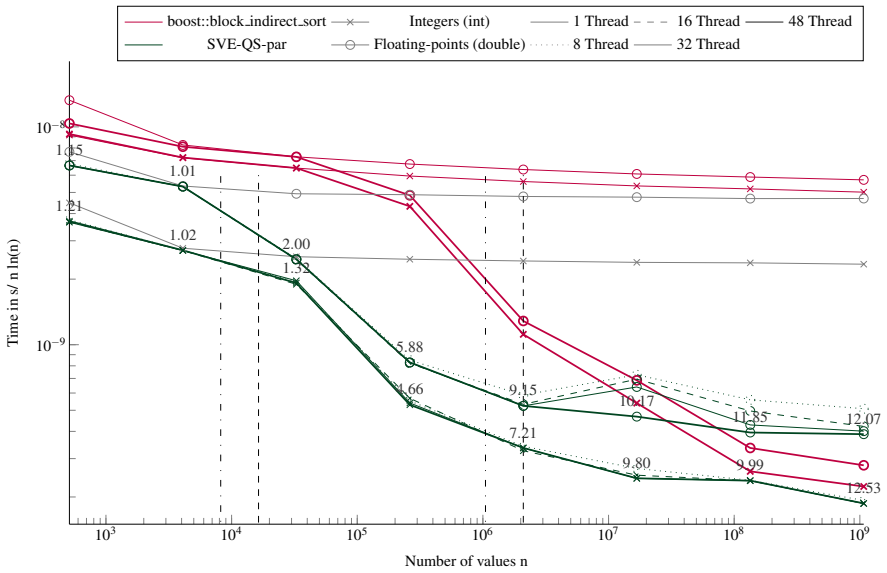


Figure 7: Execution time divided by *n ln(n)* to sort in parallel arrays filled with random values with sizes from 512 to $\approx 10^9$ elements. The execution time is obtained from the average of 5 executions with different values. The speedup of the parallel SVE-QS-par against the sequential execution is shown above the lines for 16 and 48 threads. The vertical lines represent the caches relatively to the processed data type ($-$ for the integers and $\cdot - \cdot$ for the floating-points).

but at the cost of using additional data buffers.

5.6 Comparison with the AVX-512 implementation

The results obtained in our previous study on Intel Xeon Platinum 8170 Skylake CPU at 2.10GHz [9] shows that our AVX-512-QS was sorting at a speed of $\approx 10^{-9}$ second per element. This was almost 10 times faster than the STL (10^{-8} second per element). The speedup obtained with SVE in the current study is lower and does not come from our new implementation, which is generic regarding the vector size, because the SVE512-QS is not faster. The difference does not come either from the memory accesses, because it is significant for small arrays (that fit in the L1 cache), or the number of vectorial registers, which is 32 for both hardware. Therefore, we conclude that the difference comes from the cost of the SIMD instructions or the pipelining of these.

6 Conclusions

In this paper, we described new implementations of the Bitonic sorting network and the partition algorithm that have been designed for the SVE instruction

set. These two algorithms are used in our Quicksort variant which makes it possible to have a fully vectorized implementation. Our approach shows superior performance on ARMv8.2 (A64FX) in all configurations against the GNU *C++* STL. It provides a speedup up of 5 when sorting small arrays (less than 16 SIMD-vectors), and a speedup above 4 for large arrays. We also demonstrate that an implementation designed for a fixed size of vectors is less efficient, even if this approach has good performance when implemented with AVX512 and executed on Intel Skylake. Our parallel implementation is efficient but it could be improved when working on large arrays by using a merge on sorted partitions instead of a recursive parallel strategy (at a cost of using external memory buffers). In addition, we would like to compare the performance obtained with different compilers because there are many ways to transform and optimize a C++ code with intrinsics into a binary.

Besides, these results is a good example to foster the community to revisit common problems that have kernels for x86 vectorial extensions but not for SVE yet. Indeed, as the ARM-based architecture will become available on more HPC platforms, having high-performance libraries of all domains will become critical. Moreover, some algorithms that were not competitive when implemented with x86 ISA may be easier to vectorize with SVE, thanks to the novelties it provides, and achieve high-performance. Finally, the source code of our implementation is publicly available and ready to be used and compared against.

7 Acknowledgment

This work used the Isambard 2 UK National Tier-2 HPC Service (`http://gw4.ac.uk/isambard/`) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1). In addition, this work used the Farm-SVE library [31].

A Appendix

A.1 Source code of sorting one vector of integers

In Code 1, we provide the implementation of sorting one vector using Bitonic sorting network and SVE.

```
inline void Sort1Vec(svint32_t& vecToSort){
    const int N = svcntw(); // Number of values in a vector
    const svint32_t vecIndexes = svindex_s32(0, 1); // [O, 1, ...., N-1]
    svbool_t falseTrueVecOut = svzip1_b32(svpfalse_b(),svptrue_b32()); // [F, T, ...
     F, ..., T]
    svint32_t vecIndexesPermOut = svsel_s32(falseTrueVecOut, svdup_s32(-1), ...
     svdup_s32(1));
    for(long int stepOut = 1 ; stepOut < N ; stepOut *= 2){
        {
            const svint32_t premuteIndexes = svadd_s32_z(svptrue_b32(), ...
     vecIndexes, vecIndexesPermOut);
            const svint32_t vecToSortPermuted = svtbl_s32(vecToSort, ...
     svreinterpret_u32_s32(premuteIndexes));
            vecToSort = svsel_s32(falseTrueVecOut,
                                  svmax_s32_z(svptrue_b32(), vecToSort, ...
     vecToSortPermuted),
                                  svmin_s32_z(svptrue_b32(), vecToSort, ...
     vecToSortPermuted));
        }
        svbool_t falseTrueVecIn = svuzp2_b32(falseTrueVecOut,falseTrueVecOut); // ...
     [F, F, ..., T, T]
        svint32_t vecIncrement = svdup_s32(stepOut/2);
        for(long int stepIn = stepOut/2 ; stepIn >= 1 ; stepIn/=2){
            const svint32_t premuteIndexes = svadd_s32_z(svptrue_b32(), vecIndexes,
                                svsel_s32(fftt, svsel_s32(falseTrueVecIn, ...
     svneg_s32_z(falseTrueVecIn,
                                vecIncrement), vecIncrement), vecIncrement));
            const svint32_t vecToSortPermuted = svtbl_s32(vecToSort, ...
     svreinterpret_u32_s32(premuteIndexes));
            vecToSort = svsel_s32(falseTrueVecIn,
                                  svmax_s32_z(svptrue_b32(), vecToSort, ...
     vecToSortPermuted),
                                  svmin_s32_z(svptrue_b32(), vecToSort, ...
     vecToSortPermuted));
            falseTrueVecIn = svuzp2_b32(falseTrueVecIn,falseTrueVecIn);
            vecIncrement = svdiv_n_s32_z(svptrue_b32(), vecIncrement, 2);
        }
        falseTrueVecOut = svzip1_b32(falseTrueVecOut,falseTrueVecOut);
        vecIndexesPermOut = svsel_s32(falseTrueVecOut,
                                      svsub_n_s32_z(svptrue_b32(), vecIndexesPermOut, ...
     stepOut*2),
                                      svadd_n_s32_z(svptrue_b32(), vecIndexesPermOut, ...
     stepOut*2));
    }
}

```

Code 1: Implementation of sorting a single SVE vector of integers.

A.2 Source code to check if a vector of integers is sorted

In Code 2, we provide two implementations to test if a vector is already sorted. These functions can be used if there is a chance that parts or totality of the input vector are already sorted.

```
inline bool IsSorted(const svint32_t& input){
    // Methode 1: 1 vec op, 1 comp, 2 bool vec op, 2 bool vec count
    svint32_t revinput = svrev_s32(input);
    svbool_t mask = svcmpgt_s32(svptrue_b32(), input, revinput);
    svbool_t v1100 = svbrkb_b_z(svptrue_b32(), mask);
    return svcntp_b32(svptrue_b32(),svnot_b_z(svptrue_b32(),v1100)) == ...
     svcntp_b32(svptrue_b32(),mask);
}

inline bool IsSorted(const svint32_t& input){
    // Methode 2: 1 vec op, 1 comp, 2 bool vec op, 1 bool vec count
    svbool_t FTTT = svnot_b_z(svptrue_b32(), svwhilelt_b32_s32(0, 1);
    svint32_t compactinput = svcompact_s32(FTTT, input);
    const size_t vecSizeM1 = (svcntb()/sizeof(int))-1;
    svbool_t TTTF = svwhilelt_b32_s32(0, vecSizeM1);
    svbool_t mask = svcmple_s32(svptrue_b32(), input, compactinput);
    return svcntp_b32(TTTF,mask) == vecSizeM1;
}

```

Code 2: Possible implementations to test if a vector of integers is already sorted.

References

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

[2] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a pc game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, 1998.

[3] Amir Raoofy, Roman Karlstetter, Dai Yang, Carsten Trinitis, and Martin Schulz. Time series mining at petascale performance. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 104–123, Cham, 2020. Springer International Publishing.

[4] Vaclav Snasel, Pavel Kromer, Jakub Safarik, and Jan Platos. Jpeg steganography with particle swarm optimization accelerated by avx. *Concurrency and Computation: Practice and Experience*, 32(8):e5448, 2020. e5448 cpe.5448.

[5] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[6] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, March 2017.

[7] ARM. Arm c language extensions for sve. https://developer.arm.com/documentation/100987/0000. Accessed: July 2020 (version 00bet1).

[8] ARM. Arm architecture reference manual supplement, the scalable vector extension (sve), for armv8-a. https://developer.arm.com/documentation/ddi0584/ag/. Accessed: July 2020 (version Beta).

[9] Berenger Bramas. A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *International Journal of Advanced Computer Science and Applications*, 8(10), 2017.

[10] OpenMP Architecture Review Board. Openmp application program interface, jul 2013.

[11] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

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

[13] Standard for programming language c++, iso/iec 14882:2014(e): Programming languages - c++, 2014. 25.4.1.1 sort (p. 911).

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

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

[16] David Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Computers*, 28(1):2–7, 1979.

[17] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[18] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[19] Peter M Kogge. *The architecture of pipelined computers*. CRC Press, 1981.

[20] 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.

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

[22] 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.

[23] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *European Symposium on Algorithms*, pages 784–796. Springer, 2004.

[24] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 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.

[25] Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. 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.

[26] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[27] Shay Gueron and Vlad Krasnov. Fast quicksort implementation using avx instructions. *The Computer Journal*, 59(1):83–90, 2016.

[28] K. Hou, H. Wang, and W. Feng. A framework for the automatic vectorization of parallel sort on x86-based processors. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):958–972, 2018.

[29] Z. Yin, T. Zhang, A. Müller, H. Liu, Y. Wei, B. Schmidt, and W. Liu. Efficient parallel sort on avx-512-based multi-core and many-core architectures. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 168–176, 2019.

[30] A. Watkins and O. Green. A fast and simple approach to merge and merge sort using wide vector instructions. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 37–44, 2018.

[31] Bérenger Bramas. Farm-SVE: A scalar C++ implementation of the ARM® Scalable Vector Extension (SVE), July 2020.