Quicksort Algorithm Vectorization using AVX2 on AMD Ryzen 7

Amila Hrustić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ahrustic2@etf.unsa.ba

Amila Laković
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
alakovic1@etf.unsa.ba

Samra Mujčinović
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
smujcinovi1@etf.unsa.ba

Emir Pita
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
epita1@etf.unsa.ba

***Abstract*—Today's modern CPUs are designed on hierarchical memory and SIMD/vectorization capability. This allows us to create some algorithms efficiently using new AVX (advanced vector extensions) features. This article describes a technique for implementing the Quicksort sorting algorithm and testing it with 4 different types of data. This method includes using new AVX2 instructions that were introduced with Intel's recent architecture codename Haswell on AMD Ryzen 7 8-core processor. The results are also compared with other versions of algorithms: sort, qsort and stablesort which are included in C++ *algorithm* library.**

Keywords—Quicksort, vectorization, SIMD, AVX2, sort, mask

I. Introduction

For some amount of time sorting has been a big problem in computing that has always been the subject of various researches, as it is widely used to reduce the complexity of some algorithms and procedures. It is used in a variety of applications and on a variety of platforms. Moreover, sorting is a central operation in various applications such as databases, some SQL servers [3], and even image processing.

There are a lot of sorting solutions that can be used and are based on some sorting algorithms like IPP Intel library. This library contains ready-to-use, domain-specific functions that are highly optimized for diverse Intel architecture [4]. The only problem is that this library, for sorting, uses Radix sort, and not Quicksort.

Quicksort is one of the fastest sorting algorithms that is used in practical applications. That is why this article's main focus is optimizing this algorithm. Vectorization, *i.e.* the ability to execute a single CPU instruction over multiple data (SIMD, *Single instruction, multiple data*), has continuously improved through generations of CPUs. While the difference in execution between the scalar code and its vectorized equivalent was "only" a factor of 4 in 2000 (SSE), the difference is now up to a factor of 16 (AVX-512), and a similar difference in code performance can be seen using AVX2. Therefore, it is necessary to vectorize the code to achieve high performance on modern CPUs, using special instructions and registers [1].

Given the ubiquity of AVX2 technology in home computers and even in modern smartphones, the focus and goal of this paper will be the following:

- Proposal and implementation of vectorized version of Quicksort algorithm
- Comparison of the performance of scalar and vectorized implementation of the Quicksort algorithm on the selected platform with different data types
- Analysis of the characteristics of the algorithm and the computer platform that contribute the most and the least to the acceleration of the algorithm

The key idea for this method is using a statically defined and implemented special shuffle mask that accelerates the sort operation. For this new AVX2 instructions are used [5].

The main two reference papers that use almost the same topic are: "*A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake"* by Bramas B. (2017.) and "*Fast Quicksort Implementation UsingAVX Instructions"* by Gueron S. and Krasnov V. (2015.). These two papers share the same vision of using these masks as in this paper, but the main difference between them and this is that here one and only one mask is used, and they are using several masks with lookup tables to generate them. Main idea for this paper is to use *hit-or-miss* strategy on finding *pivot* element with this one mask. The assumption is that

this strategy will be cheaper than using more masks and figuring out which one to use. Also, in both works, only one type of data was implemented and tested, while in this one it is implemented and tested on 4 different data types.

The rest of the paper is organized as follows: Section II provides basic information related to the platform and technology on which the work and testing was performed, as well as the explanation of used sorting algorithms. The approach and method of implementation of the Quicksort algorithm is presented in Section III. Finally, detailed results obtained by the implementation and the conclusion itself are presented, which can be found in Sections IV and conclusion in Section V.

II. BACKGROUND

A. Platform description

The platform that was used in this paper is *AMD Ryzen 7* processor. The selected architecture does not belong to the Intel processor family, and therefore attention is paid exclusively to the use of AVX2 technology on architecture that does not belong to the specified family.

The AMD Ryzen 7 4700U is an 8-core mobile processor, introduced in January 2020. It is part of the Ryzen 7 line, which uses the Zen 2 (Renoir) architecture with the Socket FP6. The Ryzen 7 4700U has 12 MB of L3 cache and by default runs at 1800 MHz, but can be boosted up to 4.2 GHz, depending on the load. AMD is building the Ryzen 7 4700U on a 7 nm manufacturing node using 9,800 million transistors. AMD's processor supports DDR4 memory with a dual-channel interface. The maximum officially supported memory speed is 4266 MHz, but with overclocking (and real memory modules) it can go even higher. Programs that use AVX can run on this processor, increasing performance for budget applications. In addition to AVX, AMD includes the newer AVX2 standards, but not the AVX-512 [6].

B. Technology description

The technology that this paper is based on and that was used is AVX2 (Advanced Vector Extensions 2). AVX2 are extensions of the AVX x86 architecture instruction set for Intel and AMD microprocessors proposed by Intel in March 2008. AVX supports both 128-bit SIMD instruction format and 256-bit [7].

In this paper, the goal is to develop an implementation of the Quicksort algorithm using AVX2 instructions introduced in Intel's Haswell architecture. This instruction set includes additional instructions compared to AVX, and most instructions have been extended to work with 256-bit registers. As the implementation of the parallel Quicksort algorithm was done on AMD's processor, it used the AVX2 instruction set, as the AVX-512 is only supported by Intel processors. Accordingly, it did not achieve the maximum possible sorting performance, but it is possible to achieve extremely good sorting speeds.

C. Sorting Algorithms

In this paper the goal was to compare several kinds of sorting algorithms. Firstly, there are different kinds of algorithms that were used here. C++ has a library *algorithm* that is a part of a package in all compilers that are being used [8]. That being said, this library has different kinds of sorting algorithms already implemented. In this paper, algorithms that were used are: *sort*, *qsort* and *stable_sort*. Their performances were compared with a basic Quicksort algorithm and a Vectorized one.

1. Quicksort algorithm

Quicksort is a sorting algorithm that works on the principle of *divide and conquer*. This is a very fast sorting algorithm that has a wide application. It is mostly used for educational purposes to explain algorithms and sorting, but it can also be used in practice to work with data. Its standard time complexity is $O(nlogn)$ and in the worst case is $O(n^2)$. That makes it one of the better algorithms, when it comes to speed, for sorting some large amounts of data.

As already mentioned, Quicksort uses a *divide and conquer* strategy, and it works following these three steps [9]:

- one member of the array called a *pivot* is selected first
- all elements larger than the pivot value are switched after and smaller before the pivot
- steps 1 and 2 are constantly called for subsets with smaller and larger values than the pivot, until the recursion reaches an array with size 0 or 1

Figure 1 shows how does this algorithm work:

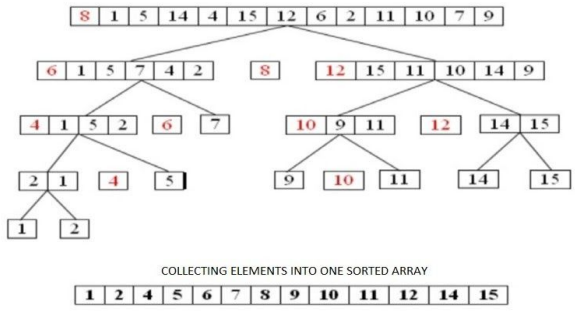


Figure 1. Quicksort algorithm example to sort [8,1,5,14,4,15,12,6, 2,11,10,7,9] to [1,2,4,5,6,7,8,9,10,11,12,14,15]. [9]

Due to the *divide and conquer* strategy, this algorithm can be easily parallelized. After dividing the string into 2 parts, these split strings can be sorted in parallel, each thread its own substring, and due to constantly calling functions the number of threads can be increased to the maximum number of threads determined in the

algorithm. This greatly reduces the time required for sorting [9].

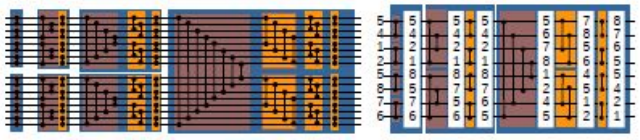
III. Implementation of quicksort with avx2

A. Underlying problem

In this section, the underlying problem is described. This problem was used to solve it in the combination of these two strategies: *divide and conquer* and *hit-or-miss*. As mentioned before, sorting is a fundamental problem in many computer science applications. The goal of this paper is to overcome the problems and performance stalls that scalar sorting algorithms face on computer architectures that support the, by now ubiquitous, AVX2 instruction set.

B. Underlying idea

The underlying idea is to vectorize the Quicksort algorithm using the AVX2 instruction set. A sorting network is an abstract description of how to sort a fixed number of values. The sorting procedure can be represented graphically, by having each array element as a horizontal line, and each compare and exchange unit as a vertical connection between those lines. There are many examples of sorting networks, but in this paper the focus is on using a bitonic sorting network. This kind of network has an algorithm complexity of $O(n \log(n)^2)$. It has been shown that it has good performances on parallel computers [10] and GPUs [11]. Figure 2 demonstrates sorting using compare and exchange units (vertical lines) on an array with 8 16-bit elements (horizontal lines) in a bitonic sorting network.



(a) A bitonic sorting network (b) Example of an array with 8 elements of size 16.

Figure 2. Bitonic sorting network example [1]. The sorting is done from left to right using compare and exchange units (shown in orange - linear progression and red - from extremities to the center).

AVX2 256-bit integer masks are used as compare and exchange units in this vectorized version of the Quicksort algorithm following the *hit-or-miss* strategy. The *hit-or-miss* strategy is used when choosing the right mask to select elements that need to be moved according to their relation to the pivot element. In this paper, only **one** static (hard-coded) **mask** is used, as opposed to many used in [1] and [2]. This mask sets the indices of elements lesser than the pivot to -1, so that they can be shuffled and sorted in the ascending order (the default sorting order in our implementation).

Because only one mask is used to select the array elements lesser than the pivot and shuffle them afterwards, the elements may not be sorted in the correct order. These kinds of mistakes are called *misses* (singular *miss*), as the masking and shuffling doesn't sort the elements in the correct order. Opposed to missed, it's possible to have *hits* (singular *hit*). Hits happen when the masking and shuffling sorts the 256-bit of elements in the correct order. To overcome this issue, the algorithm continues to iterate through the 256-bit section of the array elements and repeats the procedure on the shuffled section. This way, the cyclomatic and time overhead that is generated by multiple comparisons and function calls when selecting a mask from a pool of masks is overcomed. Mechanisms of correcting sorting faults are called *fallbacks* (singular *fallback*). By using this strategy, it's ensured that a *fallback* is available for the algorithm so that the final array comes out fully sorted.

Another fallback method that is used in this implementation of a vectorized Quicksort algorithm is a call of a scalar sorting algorithm at the end to ensure that no unsorted elements remain in the final array. As it will be discussed in the next section, the amount of elements that violate the sorted structure of the array is around 4-6% of the number of array elements. The array resulting from the basic vectorized Quicksort is, therefore, only partially sorted in a small (estimated to be between 3-9% of all cases, as discussed, again, in the Section IV. Results) number of cases. That brings us to a maximum of 0.54% of all array sorting cases that need to be corrected by the fallback sorting algorithm. Because of the semi-sorted structure of the resulting array (from the vectorized sorting phase), the scalar Quicksort algorithm as an appropriate solution is chosen. Other candidates are Insertion sort (appropriate for small numbers of elements and semi-sorted arrays) and even HeapSort.

C. Sorting with AVX2

This subsection will provide an overview of sorting with AVX2 using ideas mentioned in the previous subsection. The sorting and partitioning of one SIMD vector, sorting and partitioning of multiple SIMD vectors and small array sorting will be described. Let *VECTOR_SIZE* be the number of elements that can be aligned and fit in an AVX2 256-bit vector, e.g. VECTOR_SIZE for 32-bit integers is 8.

1) Sorting and partitioning one SIMD vector. Sorting and partitioning of one SIMD vector of *VECTOR_SIZE* elements is the basis of this Quicksort variant. The basis of partitioning one 256-bit SIMD vector of *VECTOR_SIZE* size is comparing the pivot to all elements packed in the vector and generating a mask that indicates the positions of elements greater than the pivot inside the vector (for the left and right sections of the array). Afterwards the masks are consolidated for the respective sections of the array to have the bits that indicate the position of such elements set to 1 and others to 0. The masks themselves are integers that have the size (int bits) equal to the *VECTOR_SIZE*, which means that each bit represents an index of a number contained in the vector. If the number of elements in a vector is less than 8, *e.g.* in doubles over 64-bit integers (*VECTOR_SIZE* is 4), then a 8-bit unsigned integer mask is used, to save up on memory space. When

the masks have been consolidated, they get aligned and the respective sections of the array (left and right) shuffled and merged, which is equal to sorting the initial array, or section of the array, since the sorting is done recursively. It is important to note that if the vector masks are equal to zero, then the section of the array in question is skipped and the sorting continues. This leads to sorting and partitioning multiple SIMD vectors.

2) Sorting and partitioning multiple SIMD vectors. The aforementioned principle of sorting and partitioning one SIMD vector (in the context of AVX2 ISA, one SIMD vector has a length of 256 bits) can be applied to sorting and partitioning multiple SIMD vectors. The function *partition_epi32* is a new implementation in C++ (C++11 standard) of a vectorized partitioning algorithm used in Quicksort, and, as such, represents partitioning an array in general. The algorithm starts from the extremities of the array and moves to the center, loading SIMD vectors one by one, comparing the packed numbers to the pivotal element that has been forwarded as an argument. The same algorithm for sorting one SIMD vector is used for all the SIMD vectors that are partitioned from the initial array in this manner. The merger of two 256-bit SIMD vectors into one is a process of using masks to shuffle the elements lesser than the pivot into the left section (represented by one SIMD vector) and elements greater than the pivot into the right section. If the number of elements of the pivot is lesser than VECTOR_SIZE, then a new pivot is selected from the elements (packed numbers) of both SIMD vectors and the merger is completed according to the new, temporary, pivot.

3) Sorting small arrays. Arrays smaller than *VECTOR_SIZE* are considered small arrays in the context of AVX2 SIMD instructions. These kinds of arrays can't fit in one 256-bit SIMD vector without padding or adding additional elements, which are excluded from the original set of values that the data type of the array supports. These two options are impractical, since the main goal is to sort arrays, regardless of the values of the elements it contains. No restrictions are made on which values the array may contain, other than the restrictions imposed by the lengths and format of the numerical data types. Since a signal element for padding can't be used, scalar partitioning is performed on small arrays. The time and the complexity in cycles of scalar partitioning of small arrays is insignificant, since small arrays have a maximum size of *VECTOR_SIZE-1*, which is, at most, 31 for 8-bit integers.

4) Supported numeric data types. This implementation of a vectorized Quicksort algorithm is most suitable for 32- and 64- bit integers and floating point numbers and those are the supported data types. The AVX2 ISA, along with AVX, has a wide range of intrinsic functions that support 32- and 64-bit integers and floats. Broader support for integer types was added in AVX2 [7]. Since AVX2 was not meant to generally support 8- and 16-bit integers in comparison to 32- and 64-bit integers and floating point numbers, this implementation did not include support for the former.

5) Vectorized Quicksort. The vectorized Quicksort core is similar to the scalar version of the Quicksort algorithm, shown in Figure 3.

```
template<typename NumericType>
void quicksort(NumericType *array, int left, int right) {

        int i = left;
        int j = right;

        const NumericType pivot = array[(i + j) / 2];

        scalar_partition<NumericType>(array, pivot, i, j);

        if (left < j) {
        quicksort<NumericType>(array, left, j);
        }

        if (i < right) {
        quicksort<NumericType>(array, i, right);
        }
}
```

Figure 3. Scalar implementation of Quicksort. Vectorized Quicksort core is based on this implementation.

The fallback sorting algorithm (scalar Quicksort) is called after the vectorized Quicksort. The implementation can be found on [13].

IV. Results

The results of the experiments performed using the vectorized AVX2 implementation will be demonstrated in this section. In this experiment, testing for the best, worst, and average case is performed, when the elements are already sorted in ascending, descending order, and when their order is randomized. The testing is repeated through 10 iterations and the graphs show the average value of the results of these iterations. Each of the cases are run on an array of different data types - two integers (32bit and 64bit) and two floating point types (single and double precision). Different array lengths are chosen to cover different cases of memory usage and performance, but the main focus is on arrays bigger than 8000, since the goal of this vectorized version of the Quicksort algorithm is to perform well in situations where bigger arrays are used.

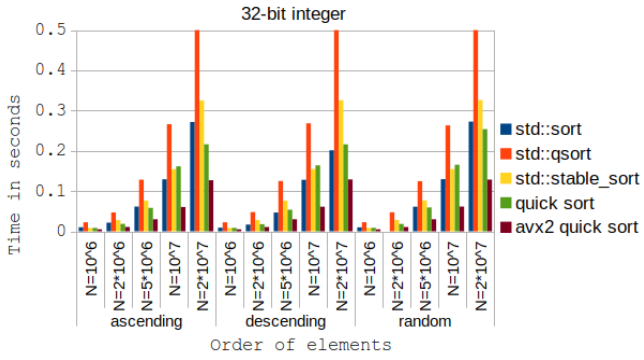


Figure 4: The time performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of 32-bit integers

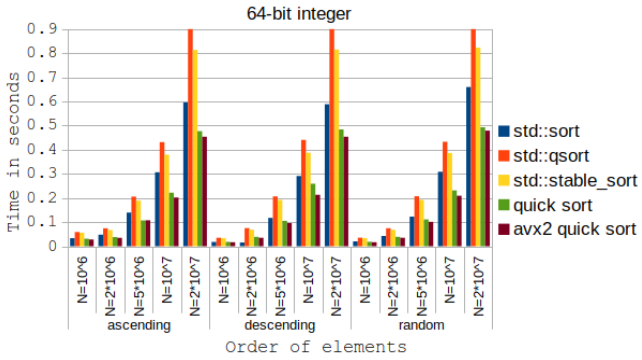


Figure 5: The time performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of 64-bit integers

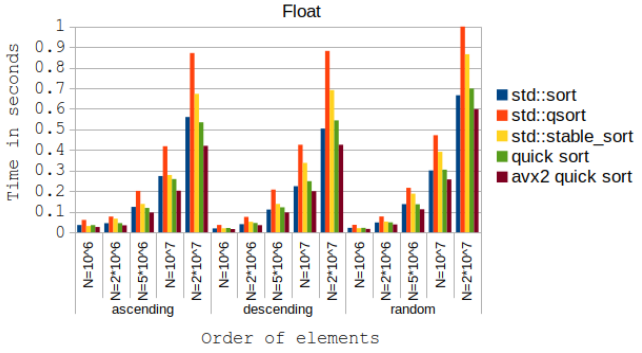


Figure 6: The time performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of single precision numbers (float)

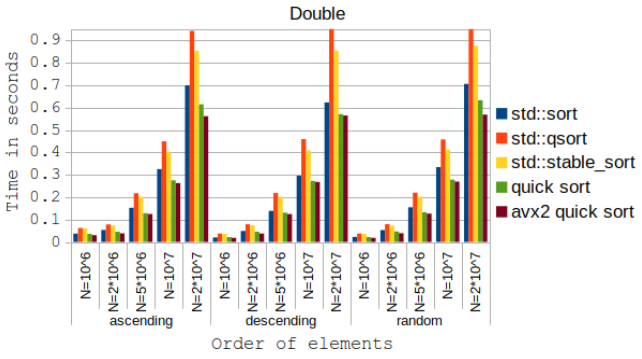


Figure 7: The time performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of double precision numbers (double)

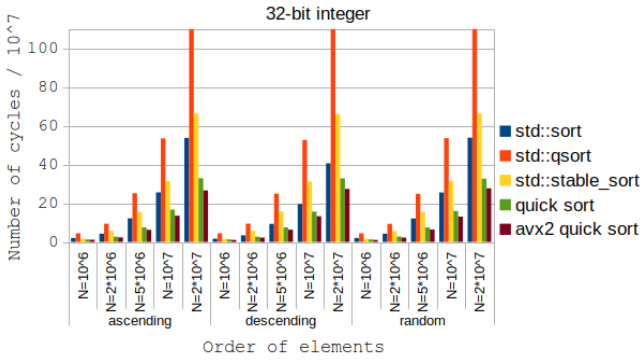


Figure 8: The complexity in cycles performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of 32-bit integers

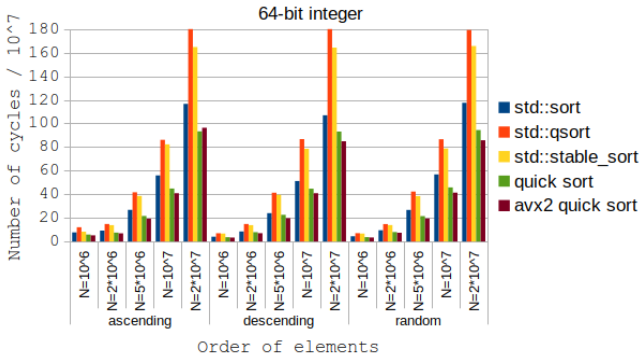


Figure 9: The complexity in cycles performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of 64-bit integers

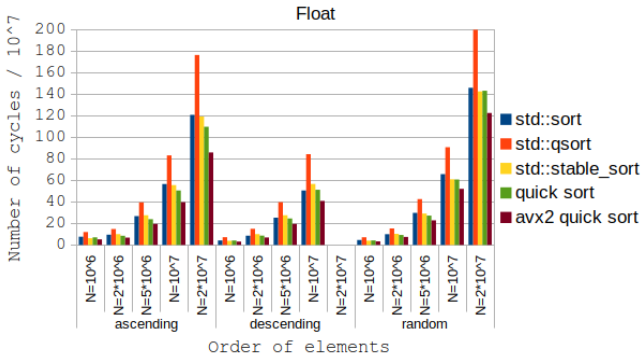


Figure 10: The complexity in cycles performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of single precision numbers (float)

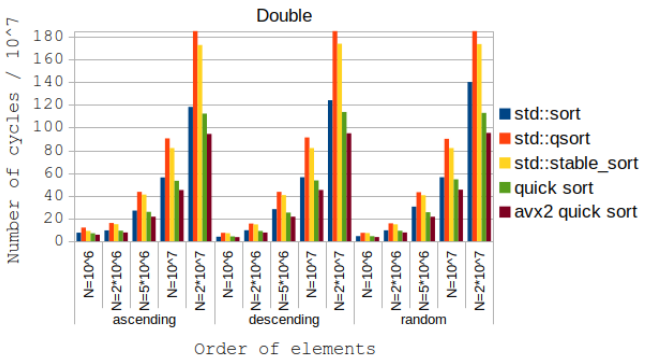


Figure 11: The complexity in cycles performance of C++ sort functions, scalar quick sort algorithm and this AVX2 quick sort algorithm tested on different size arrays of double precision numbers (double)

Figures 4-11 show the results of a series of experiments on different numeric data types and array sizes. The results are shown in seconds and other derivatives of seconds, as shown on the graphs. It is evident that the AVX2 implementation of Quicksort is faster, in terms of both time and the complexity in cycles, on average than the sorting functions found in C++ STL and the scalar implementation of Quicksort.

AVX2 vectorized Quicksort achieves best performance when sorting arrays with larger numbers of elements, *e.g.* sorting 32-bit floating point arrays with over 10 000 elements. As mentioned in Section III, a few bottlenecks are encountered when it comes to improving the performance of this proposed vectorized Quicksort.

The first issue on improving performance using this implementation is SIMD vector size. AVX2 uses 256-bit wide vectors and, therefore, this algorithm is limited by the size of the data it can compute using one instruction. This can be bypassed by using more advanced ISA, like AVX-512 which introduced 512-bit SIMD vectors, among other improvements.

Cache size can also be considered as a bottleneck, as, with arrays that can range from 10^4 to 10^6 in size, cache memory can become full and the number of cache misses would increase. The platform on which the experiments were performed has L1d 256 KB, 1i 256 KB, L2 4MB and L3 8MB caches, which are, in the average case, just enough for sorting arrays in a wide range of sizes. This implementation features a fallback mechanism that is activated when the mask misses to sort the whole SIMD vector. Since the vector elements are compared to the pivot using the *_CMP_GT_OQ* implicitly or explicitly, depending on the intrinsic function used for different types, some flags may not have been set or the signal for the comparison may not have been sent correctly. On the basis of the experiments performed, using different sizes for the array that is being sorted, it has been estimated that this happens in 3-9% of all cases. The complexity gained by using these fallback mechanisms is negligible in comparison to vectorized sorting. These experiments have shown that this happens only when sorting floating-point numbers.

V. Conclusion

In this paper, the goal was to introduce the new vectorized Quicksort algorithm implementation that leverages SIMD instructions for different data types. The performance was tested on a new Ryzen 7 processor and it showed a lot of new improvements. A new Quicksort algorithm, that is faster in an average case for all 4 different data types, has been implemented and designed. Also, when the algorithm was tested with different kinds of C++ sorting algorithms it showed better results, meaning that this vectorized algorithm was faster, especially with bigger arrays of data.

In conclusion, it has been shown that using one mask, that has been already described, can make a lot of difference in speed. This algorithm is a very stable technology and can be used in various situations with some other data types. But, the new AVX-512 instructions [12], that have been released in recent years, could benefit with this kind of algorithm. It would make this proposed solution faster, but it is not readily available on every computer architecture.

References

[1] Bramas, B. (2017). *A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake* (Vol. 8). IJACSA) International Journal of Advanced Computer Science and Applications.

[2] Gueron S., Krasnov V. (2015). *Fast Quicksort Implementation Using AVX Instructions*. The British Computer Society 2015.

[3] de Wiel, M.V. and Daer, H. (2005) Sort Performance Improvements in Oracle Database 10g Release 2, An Oracle White Paper, Oracle.

[4] Intel Integrated Performance Primitives (IPP), Intel. https://software.intel.com/en-us/intel-ipp.

[5] Buxton, N. (2011) Haswell new instruction descriptions now available. Intel. https://software.intel.com/content/www/us/en/develop/blogs/haswell-new-instruction-descriptions-now-available.html.

[6] "AMD Ryzen 7 4700U Specs." *TechPowerUp*, 25 Nov. 2020, www.techpowerup.com/cpu-specs/ryzen-7-4700u.c2282.

[7] Intel Advanced Vector Extensions (Intel AVX), Intel. http://software.intel.com/en-us/intel-isa-extensions.

[8] The GNU C++ Library, http://gcc.gnu.org/onlinedocs/libstdc++/.

[9] Šuljug, Krešimir. *Paralelne izvedbe algoritama za sortiranje*. Osijek, Croatia, 2017.

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

[11] .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.

[12] Reinders, J. (2013) AVX-512 instructions. Intel. http://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[13] Pita, Emir, Samra Mujčinović, Amila Laković, and Amila Hrustić. "QuickSortAVX.", https://github.com/emirpita/QuickSortAVX, 19 Jan. 2021. Web. 20 Jan. 2021.