An efficient Hardware implementation of TimSort and MergeSort algorithms using High Level Synthesis

Yomna Ben Jmaa*‡, Karim M. A. Ali†, David Duvivier†, Maher Ben Jemaa‡§ and Rabie Ben Atitallah†

*LAMIH UMR CNRS 8201, University of Valenciennes and Hainaut-Cambresis, France, Email: yomna.benjmaa@etu.univ-valenciennes.fr

†LAMIH UMR CNRS 8201, University of Valenciennes and Hainaut-Cambresis, France, Email:{karim.ali, david.duvivier,rabie.benatitallah}@univ-valenciennes.fr

‡ReDCAD, National School of Engineers of Sfax BP 1173-3038 Sfax, University of Sfax, Tunisia

§ Email: maher.benjemaa@enis.rnu.tn

***Abstract*—Sorting algorithms are one of the most commonly used in computer science. They can be seen as a pillar for some applications such as decision support systems, path planning, etc. However, sorting large number of elements needs high computation rate. Consequently, accelerating sorting algorithms using hardware implementation is an interesting solution to speed up computations. The purpose of this paper is to develop a hardware accelerated version of TimSort and MergeSort algorithms from high level descriptions. The algorithms are implemented using Zynq-7000 xilinx platform as part of real time decision support for avionic applications. As experimental results, we compare the performance of two algorithms in terms of execution time and resource utilization. We showed that TimSort ranges from 1.07x to 1.16x faster than MergeSort when optimized hardware is used.**

***Index Terms*—FPGA, TimSort algorithm, MergeSort algorithm, Heterogeneous architecture CPU/FPGA Zynq platform.**

I. Introduction

Nowadays, intelligent transportation systems (ITS)[1] play an important role in reducing accidents, traffic congestions and air pollution. These systems are employed in different domains such as avionics, railway and automotive. In such systems, many applications use sorting algorithms at different steps. The performance of sorting algorithms is not only affected by their complexities, but also by the targetted execution-platform(s). A set of different platforms can be used such as CPU (single or multi-core), GPU, FPGA or heterogeneous architectures. FPGAs are increasingly utilized for building complex applications at better performance measurements. These performances are achieved by profiting from the huge number of available programmable fabrics which allow to implement massively parallel architectures [2]. This huge number of logic alongside with the growing complexity of applications have driven the design methodologies to higher abstraction levels [3] and consequently, High-Level Synthesis (HLS) tools have evolved to increase the productivity of FPGA-based designs. In spite of the failure of the first HLS generations to meet the expectations of hardware designers; different reasons have encouraged researchers to enhance them for producing more efficient hardware designs. We could mention among theses reasons: the huge growth in the silicon capacity, recent designs tend to use accelerators and heterogeneous Systems on Chips (SoCs), shortening time-to-market constraint, enhancing design productivity by reusing behavioral designs instead of Register-Transfer-Level (RTL) designs, separating an algorithm from architecture to allow vast exploration for implementation alternatives [4]. Vivado HLS [5] provided by Xilinx is an example of a tool that allows to create a hardware accelerator starting from a high-level programming language (C, C++ or SystemC).

Sorting algorithm is an essential operation for real-time embedded applications. There are several classical sorting algorithms [6] such as SelectionSort, InsertionSort and BubbleSort which are easy to understand and to implement, but at low performance (complexity = $O(n^2)$). To overcome the limited performance of these algorithms [6], researchers developed MergeSort, HeapSort and QuickSort with complexity = O(n log(n)). Also, we can find some new sorting algorithms as an improvement of the previous one, for example, ShellSort appears as an improvement of InsertionSort. It subdivides the list into smaller number of sub-lists where each of which is sorted using InsertionSort. Hybrid sorting algorithms appeared as a combination of two different algorithms. For example, TimSort combines InsertionSort and MergeSort. When the number of elements is less than an optimal parameter (OP), Insertion sort is chosen; otherwise, MergeSort is used with some additional steps before the merge sorting step to ameliorate sorting execution time. This optimal parameter (OP) depends of sorting implementation and the target architecture. This work aims to develop a hardware implementation of sorting algorithms (MergeSort and TimSort) in the context of avionic applications. Further details cannot be disclosed for confidentiality reasons, however sorting is a crucial step for the underlying real time decision support methods. In order to achieve our objective, we use a High-Level Synthesis tool to generate the RTL design from behavioural description. This step requires an additional effort from the designer in order to obtain an efficient hardware implementation by using several optimization steps in three different cases: software,

978-1-5386-3250-5/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCS.2017.92

TABLE I: Sorting algorithms complexity [6]

	Best	Average	Worst
BubbleSort	O(n)	$O(n^2)$	$O(n^2)$
InsertionSort	O(n)	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Odd-EvenSort	O(n)	$O(n^2)$	$O(n^2)$
ShellSort	O(n)	$O(n(\log(n))^2)$	$O(n(\log(n))^2)$
QuickSort	O(nlog(n))	O(nlog(n)	$O(n^2)$
HeapSort	O(nlog(n))	O(nlog(n))	O(nlog(n))
MergeSort	O(nlog(n))	O(nlog(n))	O(nlog(n))
TimSort	O(n)	O(nlog(n))	O(nlog(n))

non-optimized and optimized hardware for three permutations (sorted, sorted in reverse order and random permutations). For each case and permutation a set of R replications is obtained. This constitutes our design of experiments to show how much is gained in terms of performance regarding to the hardware cost. This paper is organized as follows: Section II presents different sorting algorithms using different platforms. Section III gives a brief description of sorting algorithms and discuss how to apply the High-Level Synthesis optimization. Section IV shows experimental results. We conclude in Section V.

II. Related works

Sorting is considered as one of the well known problems in computer science. Several algorithms were proposed in the literature to solve that problem. Sorting algorithms can be classified in different ways: (i) According to algorithm complexity, Table I shows the complexity for different sorting algorithms in three different columns (best, average and worst). We can notice that four algorithms (QuickSort, HeapSort, MergeSort, timSort) have complexity of O(nlog(n)) in average column. Contrary to the other sorting algorithms, the worst case performance obtained by QuickSort is $O(n^2)$. A simple preliminary test makes all algorithms to O(n) complexity in the best case. (ii) According to the used implementation platform such as CPUs, GPUs, FPGAs and hybrid platform. Each architecture has a main specific advantage: For programmability, CPU is considered as a simple one. Whereas, FPGA is the best in terms of performance per watt. GPU appears as a solution in between. Authors in [7] used a single Intel Q9550 quad-core processor with 4GB RAM to sort up to 256M elements using MergeSort for single thread or multi-thread programs. While authors in [8] used RadixSort and MergeSort to rank 1K up to 16M elements. They executed their algorithms over different GPU platforms that scale between 4 to 30 scalar multiprocessor (SM) units. Hybrid platform SRC6 (CPU Pentium 4+FPGA virtex2) was used to implement different sorting algorithms (RadixSort, QuickSort, HeapSort, Odd-EvenSort, MergeSort, BiotonicSort) of 1000000 elements encoded in 64 bits using a single CPU and a single FPGA[9]. (iii) Selecting a sorting algorithm depends on the amount of elements. For example, InsertionSort is recommended for small number of elements. However, MergeSort is required for a large number of elements.

Several research efforts have been devoted to parallel sorting algorithms. Dominik et al [10] proposed two different hardware implementations for quick-merge parallel and hybrid algorithm. Quick-merge parallel was implemented on multicore CPU using openMP framework. Hybrid algorithm consists of parallel BiotonicSort algorithm on GPU and sequence MergeSort on CPU programmed using CUDA framework. The obtained result shows that if the number of elements is small then sorting algorithms running on GPU are faster than that running on multicore CPU. In contrast, sorting algorithms executed on multicore are more efficient for a large number of elements. They concluded that hybrid algorithm is a little bit slower than the most efficient quick-merge parallel CPU algorithm.

Davide et al [11] presented five powerful parallel sorting algorithms (MapSort, MergeSort, Tsigas-Zhangs Parallel QuickSort, Alternative QuickSort and STLSort) for sorting a large amount of data. The performance of different algorithms was tested under two different techniques: direct sorting and a key/pointer sorting. Two different machines were used: Nehalem containing 4 cores i7, 8 threads with 6 GB of memory and Westmere including 6 cores i7, 12 threads with 24 GB of memory. Davide noticed that key pointer sorting is faster than direct sorting for comparing and swapping when the size of elements is greater than the size of pointer. The algorithms were compared with respect to throughput, scalability, CPU affinity and microarchitecture analysis. Based on the obtained results, the authors recommended MergeSort and MapSort when memory is not an issue. Both these algorithms are recommended if the amount of data is up to 100000 elements as they require an intermediate array size.

Marco et al [12] proposed a high level tool for fast prototyping of parallel Divide and Conquer algorithms on multicore platforms. They implemented a pattern for multicore architecture using openMP, Intel TBB framework and FastFlow parallel programming. Their results showed that prototype parallel algorithms took the minimum time to find a solution and needed the minimum programming effort if compared to hand made parallelizations.

Bilal et al [13] proposed a new parallel algorithm called min-max butterfly network. They compared this algorithm with three other parallel sorting algorithms (Odd-EvenSort, RankSort and BiotonicSort) in terms of sorting time, sorting rate and speed up on different CPU and GPU architectures. In conclusion, BiotonicSort and Odd-EvenSort running on GPU are faster than that implemented on CPU. RankSort performs well on CPU than GPU as elements of list are dependent. They showed that the proposed parallel algorithm had a better performance than the other three sorting methods. Ajitesh et al [14] analysed a pipelined implementation of serial MergeSort network. They proposed a MergeSort based on high throughput hybrid design. The principle of this algorithm is to replace the final few stages in the MergeSort network with folded Biotonic merge networks. The hybrid design showed a throughput nearest to 10 GB/s. Konstantinos et al [15] proposed an implementation for three fundamental algorithms used in video and image processing domain. They used only the MergeSort algorithm for calculating the Kendall

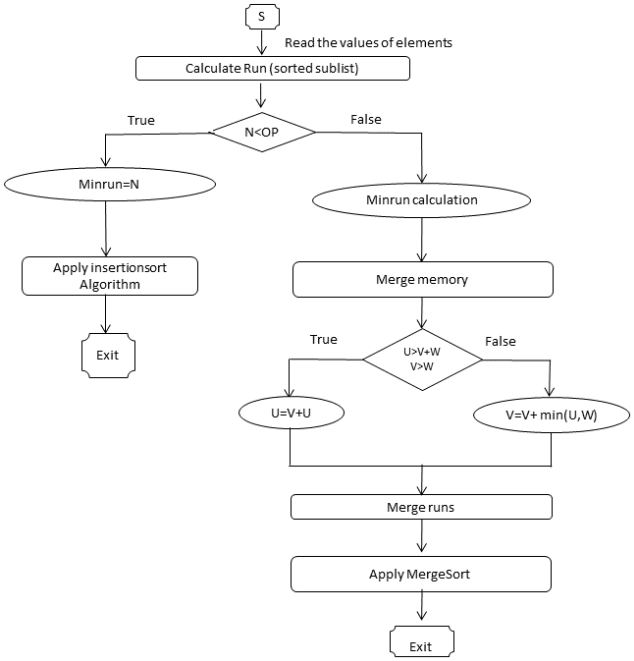

Fig. 1: TimSort algorithm

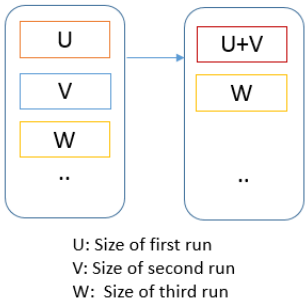

Fig. 2: Merging memory

Correlation Coefficient. Different optimisations are carried out by using different HLS directives (pipeline, unroll, array partition, inline and interface). Their results showed that hardware implementation based on virtex 7 FPGAs is 5.6x better than the software implementation. Janarbek at al [16] developed a framework that produces sorting algorithms for different criteria (speed, area, power...). Their framework contains ten basic sorting algorithms with ability to produce hybrid sorting architectures. The results showed the same performance as the existing RTL implementation for smaller arrays (<16K elements) while they overperform for large arrays (16K-130K). We are not in this context because our avionic applications need to sort at most 4096 elements issuing from previous calculation blocs.

III. HLS IMPLEMENTATIONS OF SORTING ALGORITHM

In this section, we present an overview of TimSort and MergeSort algorithms. We used Vivado HLS to generate hardware designs from behavioural description. We applied a set of optimization steps in order to obtain an efficient implementation.

A. MergeSort

MergeSort was created for the first time in 1945 by John von Neumann. It is an efficient and stable algorithm because its implementation retains the order of input to the output. MergeSort is based on the famous divide and conquer paradigm. Firstly, array is splitted into two sub-arrays, secondly, these two arrays are sorted recursively and finally the two sorted arrays are merged to obtain the result.

B. TimSort

TimSort is a combination of binary InsertionSort and MergeSort. It is a stable and adaptive algorithm. The functionality of this algorithm is based on the optimal parameter (OP) for switching between two algorithms. The value of this parameter is fixed to 64 for a sequential architecture (Processor Intel i7). For parallel architecture, we use different values for this parameter and we note that the execution times are almost similar. So, this parameter is set to 64 in the following work because it is the standardized value used by other authors but also in order to allow cross-comparisons. Based on the number of elements to be sorted, we could follow two different ways. If the size of array is less than 64 elements then InsertionSort will be selected; otherwise, MergeSort is considered in the sorting step.

Figure 1 shows the steps for applying TimSort algorithm:

(1) **Run calculation**: The input arrays may have some elements that are already in order, so we can benefit from this natural order to simplify the sorting process. These natural ordered elements are extracted in temporary arrays called *run*.

(2) **Algorithm selection step**: InsertionSort will be selected if N is less than 64; otherwise, MergeSort is considered (where N is the size of array).

(3) **Minrun calculation**: Minrun is the minimum length of runs that depends on the length of the input array. Minrun is equal to N if InsertionSort algorithm is selected but it is equal to $\frac{N}{Minrun} < 2^N$ if MergeSort is selected.

(4) **Merge memory**: These runs need to be combined to obtain the final ordered array. TimSort merges only the consecutive runs. Figure 2 represents three lengthes of runs (U,V,W) where the two conditions have to be satisfied: (a) U $>$ V+W; (b) V $>$ W. If the first rule is not satisfied then, V is merged with the minimum of U or W. This step is repeated until conditions (a) and (b) are satisfied.

(5) **Merge run**: Two adjacent runs are merged by applying the following steps:

(i) The elements of the minimum size run are copied to a temporary run.

(ii) The elements of the largest run are compared to the temporary run where the smallest value is moved in a

```
1  #define size N
2  struct out_axis
3  {
4   int data[size]
5   int TLast[size]
6  };
7
8  Function Mergesort(int *input, struct out_axis *
       result)
9  {
10  # pragma HLS INTERFACE s_axilite port=return
11  # pragma HLS INTERFACE axis port=input
12  # pragma HLS INTERFACE axis port=result
13  int array[size];
14  int i, z;
15  mergesort_label10:for(int i=0; i<size; i++)
16  {
17   #pragma HLS UNROLL factor=2
18   #pragma HLS PIPELINE
19   array[i]=input[i];
20  }
21  mergesort_label11:for(int c=1; c<=size-1; c=2*c)
22  {
23   #pragma HLS PIPELINE
24   mergesort_label12:for(int l=0; l<size-1; l=l+2*c
         )
25   {
26    #pragma HLS UNROLL factor=2
27    #pragma HLS PIPELINE
28    int mid=l+c-1;
29    int r=min(l+2*c-1, size-1);
30    merge(array, l,mid, r);
31   }
32  }
33  mergesort_label13:for(z=0; z<size; z++)
34  {
35   #pragma HLS UNROLL factor=2
36   #pragma HLS PIPELINE
37   if(z<size-1)
38   {
39    result->data[z]=array[z];
40    result->TLast[size-1]=0;
41   }
42   else if (z==size-1)
43   {
44    result->data[size-1]=array[size-1];
45    result->TLast[size-1] = 1;
46   }
47  }
48 }
```

Listing 1: Top level function for MergeSort algorithm

new array. The pointer is updated for the array from which the element was taken.

(iii) Repeat the previous steps till the end of one of the arrays.

(6) **MergeSort**: MergeSort is applied to sort the rest of elements in the array.

MergeSort is naturally a recursive algorithm. Vivado HLS cannot convert recursive functions into hardware design; therefore, we have to rewrite them in iterative way. In order to have an efficient hardware implementation for TimSort and MergeSort, we applied several optimisation steps to the C code. These steps could be detailed as following:

(1) **Loop unrolling**: Array elements are stored in BRAM memory structures which are characterized by dual physical ports. The dual ports could be configured as (dual read ports, dual write ports or one port for each operation). We benefit from this by unrolling the loops in the design by factor=2. For example, writing elements are only executed in the loop listed in Listing 1, Line 17. Therefore, the two ports for array[] could be configured as writing ports and consequently we could unroll the loop by factor=2. The same apply for the loops at Listing 1, Lines 26, 35.

(2) **Loop pipelining**: Loop iterations are pipelined in order to decrease the overall execution time. In our design, loop iterations are pipelined with only one clock cycle difference in-between by applying loop pipelining with Interval iteration (II)=1. The tool will schedule loop execution to satisfy that condition as listed in Listing 1, Lines 18, 23, 27, 36.

(3) **Input/output Interface**: For data transfer, input/output ports are configured to use AXI-Stream protocol with minimum communication signals (DATA, VALID and READY). While AXI-Lite protocol is used for design configuration purposes; for example, to identify the system current state (start, ready, busy) as in Listing 1, Lines 10-12.

IV. Experimental Results

In this section, we present our results of execution time and resource utilization for sorting algorithms on hardware architecture. We compare our results for different cases: Software version, non-optimized and optimized hardware for three permutations (sorted, sorted in reverse order and random permutations). For each case and permutation a set of R=50 replications is obtained. A set of 50 replications is called a scenario in the following paragraphs. The array size ranges from 8 to 4096 integers encoded in 4 bytes. As previously mentioned, we limited the size to 4096 elements because our sorting algorithms are mainly used for real time decision support system for avionic applications. In this case, it sorted at most 4096 actions issuing from the previous calculation blocks. We develop our hardware implementation for TimSort and MergeSort using Zedboard (XC7Z020CLG484). The hardware architecture was synthesized using Vivado suite 2015.4 with default synthesis/implementation strategies.

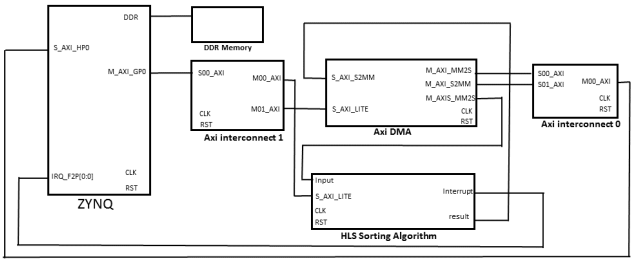

Fig. 3: System architecture block design

Figure 3 shows how the HLS IP (*HLS_sorting*) is connected to the design. The elements to be sorted are stored in the memory of the processing element (*DDR_Memory*). They are transferred from the processing sytem to the HLS core through AXI-DMA communication. After data are sorted, the result is written back through the reverse path.

TABLE II: Execution time of TimSort for software, non-optimized and optimized hardware

	Software (us)			Non-Optimized Hardware (us)			Optimized Hardware (us)		
Size of arrays	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
8	1.14	1.18	2.31	11.85	18.75	25.02	11.63	18.53	24.79
16	4.05	4.09	5.07	14.36	21.25	27.52	14.21	21.11	27.38
32	14.29	14.37	15.42	22.08	28.98	35.25	22.10	29	35.27
64	45.85	46.32	56.89	53.1	60.04	66.3	43.94	50.84	57.11
128	102.38	103.04	114.01	111.02	118.05	124.32	89.35	96.36	102.63
256	219.95	220.07	228.44	230.5	240.8	245.12	190.15	197.22	203.49
512	552.62	553.82	565.81	548.23	555.24	561.51	412.99	419.93	426.2
1024	1202.41	1205.97	1220	1230.53	1237.47	1243.74	923.10	930.5	936.32
2048	2623.61	2628.01	2642.43	2680.94	2687.88	2694.15	1989.95	1996.84	2003.11
4096	5703.5	5752	5741.88	5843.43	5850.25	5856.44	4317.44	4324.25	4330.44

TABLE III: Execution time of MergeSort for Software, non-optimized and optimized hardware

	Software (us)			Non-Optimized Hardware (us)			Optimized Hardware (us)		
Size of arrays	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
8	3.3	3.36	5.43	14.03	20.92	27.19	13.73	20.62	26.89
16	8.31	8.41	11.39	18.97	25.86	32.13	18.76	25.65	31.92
32	19.82	19.96	24.5	30.04	36.95	43.22	29.41	36.3	42.56
64	47.91	48.03	53.19	54.42	61.36	67.6	53.23	60.14	66.41
128	103.18	103.46	109.96	108.54	115.44	121.7	108.42	115.33	121.6
256	229.66	230.03	236.3	226.98	233.87	240.14	221.94	228.83	235.1
512	584.28	606.05	641.14	484.14	491.04	497.32	474.06	480.96	487.23
1024	1298.01	1298.16	1305.08	1039.7	1046.77	1053.04	1039.43	1046.33	1052.6
2048	2850.97	3005.76	3116.94	2232.46	2239.48	2245.75	2191.32	2198.22	2204.48
4096	5766.49	5916	6132.47	4782.3	4789.15	4795.34	4658.56	4665.4	4671.59

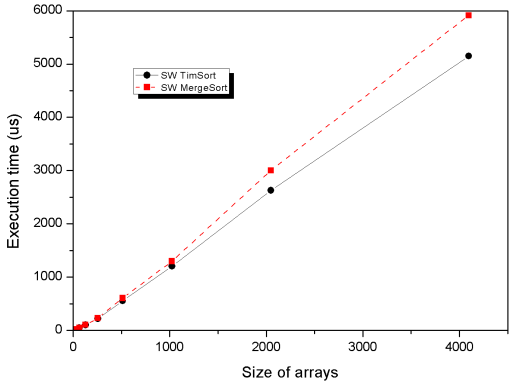

Fig. 4: Execution time of TimSort and MergeSort on ARM Cortex A9

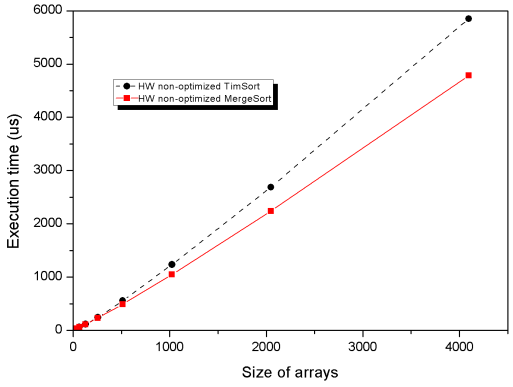

Fig. 5: Execution time of non-optimized hardware implementation of TimSort and MergeSort

TimSort and MergesSort were executed on the processing system (ARM Cortex A9 processor) to show how much the algorithm is accelerated using FPGA. We calculated the execution time of TimSort and MergeSort for different sizes of arrays ranging from 8 to 4096 elements in each scenario with 50 replications (R=50).

- **Scenario 1: Unsorted Data**

The elements are randomly generated (not sorted). Table II and Table III reported the minimum, average and maximum execution time for each case.

Figure 4 shows the execution time of TimSort and MergeSort algorithms for the average cases. For example, when N=4096, the execution time was 5916 us and 5752 us for MergeSort and TimSort respectively. For large number of elements, we concluded that TimSort is 1.17x faster than MergeSort running on ARM Cortex A9 processor (Figure 4).

Figure 5 shows the execution time of TimSort and MergeSort for non-optimized hardware implementation. When N is greater than 64, MergeSort was 1.03x-1.19x faster than TimSort; otherwise, TimSort was 1.1x-1.22x faster. For example, when N=2048, the execution time was improved by 16%. We used HLS directives in order to improve the performance. For example, Fig. 6 depicted that the optimized hardware for TimSort is faster than that for MergeSort by 12% while

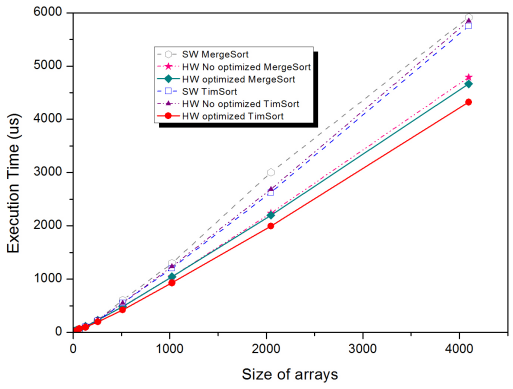

Fig. 6: Comparison between TimSort and MergeSort algorithms for unsorted data (scenario 1)

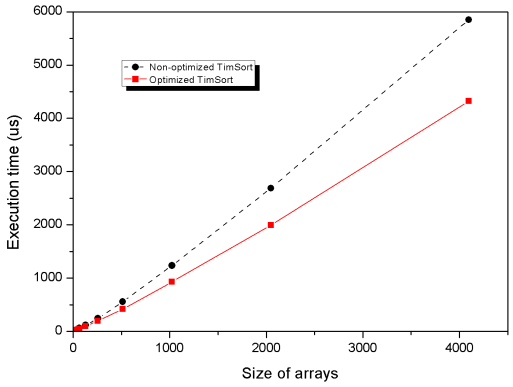

Fig. 7: Comparison between non-optimized TimSort algorithm and optimized TimSort

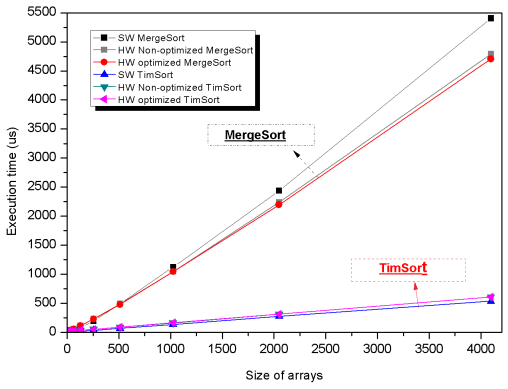

Fig. 8: Comparison between TimSort and MergeSort algorithms for ascending order of elements (scenario 2)

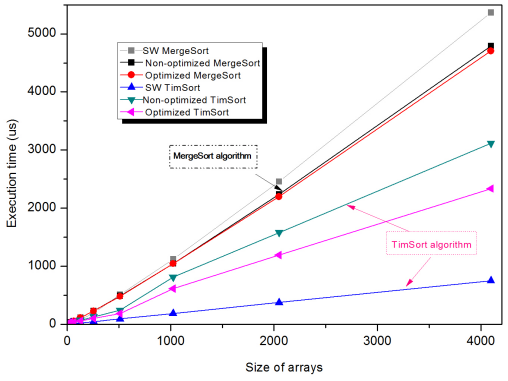

Fig. 9: Comparison between TimSort and MergeSort algorithms for descending order of elements (scenario 3)

Figure 7 showed that it was 25% faster than the non-optimized hardware implementation.

Moreover, we notice that the computational time in hardware implementation is reduced compared to software if the number of elements is important. For example, when N=1024, the hardware implementation was 1.20x-1.22x faster than software implementation for TimSort and MergeSort (Table II and Table III).

- **Scenarios 2 et 3: Sorted Data**

  The value of elements in these scenarios are sorted by default:

  – **Scenario 2 - Ascending order**:
  The values of elements are sorted in ascending order. We show the minimum, average and maximum execution time for MergeSort and TimSort algorithms on each case (Table IV and Table V). In figure 8, we note that the value of execution time for non-optimized and optimized TimSort algorithm are nearly equal. TimSort was 1.66x-1.90x and 1.13x-1.87x faster than MergeSort running on ARM Cortex A9 processor and Hardware architecture FPGA respectively. For example, when the number of elements is set to 4096, the time was 606.8 us for no optimized and optimized TimSort, 4789 us and 4707 us for non-optimized and optimized MergeSort.

  – **Scenario 3 - Descending (Reverse) order**:
  The values of elements are sorted in descending order. Table VI and Table VII present the execution time for MergeSort and TimSort algorithms for each case (Software, non optimized and optimized hardware). Figure 9 shows that TimSort was 1.33x-1.85x faster than MergeSort executing on ARM Cortex 9 and it was 1.08x-1.36x and 1.08x-1.5x faster than MergeSort running on FPGA architecture with non-optimized and optimized algorithm respectively.

TABLE IV: Execution time of MergeSort for Software, non-optimized and optimized hardware using ascending order of elements (scenario 2)

	SW (us)			Non-Optimized Hardware (us)			Optimized Hardware (us)		
Size of arrays	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
8	3.25	3.27	4.84	14.23	21.16	27.43	14.03	20.97	27.24
16	7.78	7.82	9.61	18.92	25.84	32.11	18.84	25.74	32.01
32	17.89	17.93	19.82	30.24	37.20	43.47	29.48	36.37	42.64
64	40.06	40.14	42.12	54.72	61.68	67.95	53.19	60.10	60.37
128	89.09	89.19	91.01	108.69	115.65	121.92	108.58	115.49	121.76
256	194.71	194.79	196.84	226.8	233.76	240.03	222.08	226.98	235.24
512	486.34	487.69	492.89	484.05	491	497.27	474.06	479.99	487.26
1024	1125.37	1126.8	1130.92	1039.4	1046.62	1052.62	1039.84	1045.8	1053.07
2048	2433.22	2437.19	2447.34	2234.44	2239.37	2245.64	2191.24	2198.17	2204.44
4096	5259.92	5404.85	5424.7	4782.46	4789.33	4795.52	4700	4707.23	4713.43

TABLE V: Execution time of TimSort for Software, non-optimized and optimized hardware using ascending order of elements (scenario 2)

	SW (us)			Non-Optimized Hardware (us)			Optimized Hardware (us)		
Size of arrays	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
8	1.03	1.10	2.02	11.5	18.4	24.6	11.5	18.4	24.6
16	1.99	2.04	2.88	12.75	19.64	25.9	12.75	19.6	25.85
32	4.11	4.16	4.89	15.09	22.25	28.52	15.09	22.25	28.5
64	9.14	9.58	12.27	20.1	27.09	33.36	20.08	27.04	33.31
128	17.43	18.17	20.74	29.39	36.35	42.75	29.3	36.35	42.72
256	34.38	35.63	36.55	47.87	54.8	61.5	47.8	54.75	61.2
512	69.39	69.51	71.73	84.95	91.95	97.73	84.85	91.81	97.7
1024	138.09	138.84	139.73	158.3	165.45	171.6	158.2	165.32	171.59
2048	260.52	276.62	277.79	305.49	312.45	318.72	305.22	312.41	318.68
4096	525.92	536.42	537.75	599.86	606.81	613.08	599.87	606.84	613.11

TABLE VI: Execution time of MergeSort for Software, non-optimized and optimized hardware using descending order of elements (scenario 3)

	SW (us)			Non-Optimized Hardware (us)			Optimized Hardware (us)		
Size of arrays	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
8	3.24	3.27	4.8	14.11	21.02	27.29	14.09	21	27.2
16	7.71	7.74	9.26	19.10	26	32.27	18.71	25.6	31.87
32	17.89	17.92	19.41	30.12	37.08	43.35	29.34	36.26	42.53
64	40.25	40.31	41.52	54.55	61.51	67.78	53.22	60.11	66.38
128	89.23	89.34	90.79	108.74	115.67	121.94	108.42	115.31	121.58
256	193.61	193.95	195.9	226.75	233.68	239.95	222.05	228.95	235.22
512	512.81	513.04	514.58	484.59	491.51	497.79	474.35	481.32	487.59
1024	1121.3	1122.11	1127.01	1039.5	1046.46	1052.73	1039.39	1046.35	1052.6
2048	2455.75	2456.01	2463.14	2232.65	2239.61	2245.88	2192.63	2198.58	2204.85
4096	5282.32	5365.56	5384.27	4782.35	4789.24	4795.44	4700.38	4707,28	4713.47

TABLE VII: Execution time of TimSort for Software, non-optimized and optimized hardware using descending order of elements (scenario 3)

	SW (us)			Non-Optimized Hardware (us)			Optimized Hardware (us)		
Size of arrays	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
8	2.13	2.16	3.35	12.45	19.35	25.62	12.4	19.3	25.6
16	7.09	7.14	8.21	17.11	24.02	30.29	17.1	24	30.2
32	22.39	22.47	23.78	32.30	39.26	45.53	32.3	39.26	45.53
64	12.58	12.69	15.45	38.91	45.88	52.15	34.09	41.05	47.32
128	23.94	24.43	27.23	66.65	73.62	79.89	55.08	62.01	68.28
256	48.18	48.48	49.76	125.91	132.88	139.44	95.91	102.81	109.14
512	94.64	94.75	97.17	237.47	244.43	250.7	178.96	185.91	192.19
1024	182.56	189.21	189.75	806.18	813.14	819.41	609.18	616.15	622.41
2048	361.52	377.62	378.41	1573.48	1580.45	1586.7	1186.7	1193.66	1199.93
4096	720.5	752.8	754.61	3110.43	3117.39	3123.66	2330.23	2337.19	2343.46

In the studied case, the execution time of optimized hardware implementation is 1.02x-1.13x faster than software implementation for MergeSort algorithm when N>256 (Table IV and Table VI).

From scenarios 2 and 3, it is observed that the results obtained for TimSort in scenarios 2 and 3 have a less execution time than scenario 1 in each case because it decreases the number of instructions to be executed. Thus, the performance

TABLE VIII: Resource utilization of TimSort and MergeSort

	Without optimization						**With optimization**					
	MergeSort			TimSort			MergeSort			TimSort		
	LUT	FF	BRAM-36K	LUT	FF	BRAM-36K	LUT	FF	BRAM-36K	LUT	FF	BRAM-36K
8	936	863	0	180	121	0	955	877	0	187	131	0
16	944	866	0	186	130	0	985	883	0	191	141	0
32	830	789	1.5	131	105	0.5	856	809	1.5	151	119	0.5
64	837	792	1.5	3217	2457	34.5	871	815	1.5	3993	2703	34.5
128	849	795	1.5	3247	2472	34.5	885	821	1.5	3655	2722	34.5
256	856	798	1.5	3289	2490	34.5	893	827	1.5	4068	2741	34.5
512	884	817	1.5	3308	2502	34.5	925	849	1.5	4097	2760	34.5
1024	895	820	3	3328	2517	34.5	936	855	3	4125	2779	34.5
2048	942	861	6	3389	2532	35.5	980	890	6	4182	2800	35.5
4096	918	826	12	3419	2548	37.5	954	867	12	4223	2822	37.5

of sorting algorithms depends on the permutation.

Table VIII shows the different resource utilization for TimSort and MergeSort. For TimSort, when N < 64, less resources were used since InsertionSort was systematically selected. When N$>=$ 64, TimSort shows 25% gain in execution time at 25% increase for Slice LUT and around 10.5% for Slice Register. For MergeSort, The optimized hardware consumed is between 2-5% more for both Slice Register and Slice LUT to gain between 1.5-3.5% in performance.

V. Conclusion

We presented in this paper the hardware implementation for both TimSort and MergeSort with different number of elements encoded in 4 bytes. We used High-Level Synthesis tool to generate the RTL design from behavioural description.

Comparative analysis of different results obtained by these sorting algorithms was done based on three different cases (software: ARM Cortex A9, non-optimized and optimized hardware) and two parameters (Execution time and resource utilisation) on Zynq-7000 platform. From the results, it is concluded that the TimSort ranges from 1.07x-1.16x faster than MergeSort when using optimized hardware using unsorted data.

As future work, we plan to use the hardware Timsort algorithm in avionic domain and decisional system with floating point numbers encoded in 8 bytes. The next step consists in swithing from mono-criterion to multi-criteria sorting algorithms required in the targeted avionic functions. In addition, we plan to include the best solution in hardware path planning algorithms.

References

[1] L. Figueiredo, I. Jesus, J. T. Machado, J. Ferreira, and J. M. de Carvalho, "Towards the development of intelligent transportation systems," in *Intelligent transportation systems*, 2001.

[2] M. Baklouti, Y. Aydi, P. Marquet, J. Dekeyser, and M. Abid, "Scalable mpNoC for massively parallel systems - Design and implementation on FPGA," *Journal of Systems Architecture*, 2010.

[3] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Design Test of Computers*, July 2009.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, April 2011.

[5] *Vivado Design Suite User Guide High-Level Synthesis*, Xilinx, November 2015.

[6] A. K. Karunanithi *et al.*, "A survey, discussion and comparison of sorting algorithms," *Department of Computing Science, Umea University*, 2014.

[7] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proceedings of the VLDB Endowment*, 2008.

[8] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Parallel & Distributed Processing, IPDPS, IEEE International Symposium on*. IEEE, 2009.

[9] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang, "Performance of sorting algorithms on the SRC 6 reconfigurable computer," in *Proceedings. IEEE International Conference on Field-Programmable Technology*, 2005.

[10] D. Zurek, M. Pietron, M. Wielgosz, and K. Wiatr, "The comparison of parallel sorting algorithms implemented on different hardware platforms," *Computer Science*, 2013.

[11] D. Pasetto and A. Akhriev, "A comparative study of parallel sort algorithms," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. New York, USA: ACM, 2011.

[12] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati, "A divide-and-conquer parallel pattern implementation for multicores," in *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*. ACM, 2016.

[13] B. Jan, B. Montrucchio, C. Ragusa, F. G. Khan, and O. Khan, "Fast parallel sorting algorithms on GPUs," *International Journal of Distributed and Parallel Systems*, 2012.

[14] A. Srivastava, R. Chen, V. K. Prasanna, and C. Chelmis, "A hybrid design for high performance large-scale sorting on FPGA," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015.

[15] K. Georgopoulos, G. Chrysos, P. Malakonakis, A. Nikitakis, N. Tampouratzis, A. Dollas, D. Pnevmatikatos, and Y. Papaefstathiou, "An evaluation of vivado HLS for efficient system design," in *ELMAR, 2016 International Symposium*. IEEE, 2016.

[16] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016.