ISSN 2007-9737

Sorting Algorithms Comparison on FPGA and Intel i7 Architectures

Yomna Ben-Jmaa[1,*], David Duvivier[2]

[1] University of Sfax, ReDCAD Laboratory, Tunisia

[2] Polytechnic University of Hauts-de-France, LAMIH Laboratory, France

yomna.benjmaa@redcad.org, david.duvivier@uphf.fr

Abstract. Sorting is an essential operation in many real-time applications, and choosing the right architecture to perform sorting tasks can significantly impact performance. This study aims to provide a comprehensive literature review on the implementation of several sorting algorithms on Intel i7 and FPGA architectures. On these architectures, we analyze and compare the performance and temporal stability of five different sorting algorithms: quick-sort, heap-sort, shell-sort, merge-sort, and tim-sort. Their performance are evaluated in terms of average and standard deviation of computational times on different numbers of elements ranging from 8 to 4096. The maximum number of elements to be sorted is set to 4096, as this is the number provided for a real-time decision support system as solutions to be sorted. However, our study provides insights into the performance of different sorting algorithms on different architectures, which can be useful for selecting the appropriate architecture for real-time computing applications in decision support systems.

Keywords. Field programmable gate array (FPGA), computational times, sorting algorithm.

1 Introduction

Nowadays, Embedded electronic systems have become ubiquitous in various sectors of activity, from transportation to healthcare, from consumer electronics to industrial automation.

One of the primary objectives of designing embedded electronic systems is to ensure their reliability and cost-effectiveness while performing complex tasks that meet the constraints of time, energy consumption, and manufacturing cost. Focusing on the scope of this paper, it is possible to mention as an example that flight plan planning algorithms used to find the shortest route or propose real-time avoidance trajectories incorporate sorting algorithms.

Indeed, sorting algorithms [29, 43, 1, 2] (sorts for short) play a vital role in the design and performance of embedded systems. In these systems, sorts are designed to meet the constraints of time and energy consumption. The choice of a sort depends on the nature of the data, the constraints, and the desired performance.

For example, bubble-sort or insertion-sort algorithms are easy to implement but are not suitable for large datasets or real-time applications. Conversely, quick-sort [37, 19], heap-sort [39], tim-sort [6, 22] and merge-sort [31, 19] algorithms are more complex and require more processing power but can handle large datasets and real-time applications. As technology continues to evolve, the demand for efficient and reliable embedded systems will only increase, and sorts will continue to play a vital role in meeting these demands.

Among these systems, FPGAs (Field-Programmable Gate Array) have emerged as an interesting alternative to accelerate software applications. FPGAs are integrated circuits that can be programmed to perform specific functions. They offer high performance and energy efficiency, making them suitable for real-time computing applications. Considering sorting operations, one approach is to use the FPGA's built-in hardware resources to implement the sorts directly.

These algorithms can be optimized for the FPGA architecture, resulting in faster and more efficient sorting. Another approach is to use the FPGA to offload sorting operations from the CPU. In this approach, the CPU sends the data to be sorted to the FPGA, which sorts and returns the sorted data to the CPU.

This approach can significantly reduce the processing time required for sorting, as the FPGA can handle large data sets in parallel. One challenge in using FPGAs is the programming complexity since FPGA programming requires specialized knowledge in HDLs (Hardware Description Languages) such as Verilog and VHDL (very-high-speed integrated circuit hardware description language).

However, available tools and libraries can ease the programming process, making it more accessible to software developers. Moreover, CPUs can certainly be used in embedded systems. In many cases, embedded systems require a combination of processing power and low power consumption, which can be challenging to achieve with CPUs alone. To meet these requirements, CPUs are often used in conjunction with other components, such as FPGA's microcontrollers, memories, communication interfaces, sensors, and other peripherals.

These components are integrated into a complete embedded system, which is designed to meet the specific needs of the application. The choice between CPUs and FPGAs really depends on the specific needs of the application. CPUs are great for applications that require flexibility and versatility, as they can be programmed to handle a wide range of tasks. On the other hand, FPGAs are designed for specific tasks, and are optimized for performance in those tasks.

This makes them ideal for applications that require high performance. While they may be more expensive to produce and harder to program than CPUs, their performance benefits can make them the best choice for some applications.

The objective of this paper is to compare optimized hardware and software implementations of heap-sort, shell-sort, quick-sort, tim-sort and merge-sort on FPGA and Intel i7 architectures using a limited number of elements ranging from 8 to 4096. Indeed, contrary to most of related works in the literature, there are 4096 elements at most because they are provided to a real-time decision support system as solutions to be sorted.

This is highlighted by the work of K. Nikolajevic [35], whose thesis aims to tackle the challenging problem of reducing operational accidents in avionics systems. As part of the collision avoidance alarm system, sorting these solutions efficiently is crucial for real-time decision making in avionics systems to select the best actions to avoid accidents.

To be more precise, each solution (i.e. a short-term path to follow in an avionics application) is identified by a unique 32-bit integer, so-called index, and evaluated on the basis of various performance criteria (such as distance...). Consequently, index-sorts are used in the real-life application. However, sorted elements constitute permutations of integers in this paper to simplify the problem. In general terms, the main contributions of this paper are as follows:

– Analysis and comparisons of the performance of five sorts: quick-sort, heap-sort, shell-sort, merge-sort, and tim-sort. Their performances are evaluated in terms of average and standard deviation of computational times on Intel i7 and FPGA architectures. These measurements are refined by statistical tests.

– Temporal stability analysis of the sorts: In addition to ranking the performance of the sorts on each platform, a statistical analysis is carried out on the basis of "boxplot-like" statistical measurements to assess the temporal stability of these algorithms.

The paper is structured as follows: Section 2 presents a state of the art on various sorts and several applications using different platforms (CPU, FPGA). Section 3 shows our experimental results. Section 4 gathers our conclusions and some of our future works.

2 Related Works

Over the past few decades, there has been a significant amount of research conducted on sorts. While many of these studies have focused on accelerating sorts in heterogeneous computing systems, there has also been a focus on reducing computational time, power consumption, and hardware resources. This literature review section provides an overview of numerous studies on both FPGA-based and CPU-based sorts that aim to improve acceleration performance.

2.1 Hardware Acceleration Methods for Embedded Systems (FPGA)

FPGAs are greatly flexible and customizable to meet the specific requirements of different applications. This flexibility allows FPGAs to be optimized either to perform high-performance parallel processing and data streaming in applications that require high throughput, or for fast response times in applications that require low latency, or finally for energy efficiency in applications that require low-power consumption.

The traditional development of FPGA-based applications has been mainly based on highly specialized register transfer level (RTL) designs [44, 18, 42, 5, 13, 34, 20]. High-Level Synthesis (HLS) allows increasing design productivity and detaching the algorithm from architecture [16]. In our case, the vivado HLS tool is used to generate hardware accelerators from C language.

Ben Jmaa et al. [9] proposed an efficient hardware implementation for different sorts using high-level descriptions in a zynq-7000 platform. The authors compared the performance of the algorithms in terms of computational time, standard deviation and resource utilization. The results showed that the selection-sort was 1.01-1.23 times faster than other algorithms for less than 64 elements; otherwise, tim-sort was the best algorithm. Kobayashi et al. [26, 27] detailed a new approach to reducing FPGA programming costs while maintaining high levels of performance.

Their sorting library can use OpenCL for FPGA. This approach consumed at least twice the hardware resources of the merge-sort method restructured for the OpenCL programming model for FPGA. However, it operated at a frequency 1.08 times higher and had a sorting throughput three orders of magnitude greater than the baseline.

Chen et al. [14] proposed a sample-sort algorithm on a server with a PCIe-connected FPGA to sort large data sets. The prototype system was implemented using Verilog HDL on Amazon Web Services (AWS) FPGA instances equipped with Xilinx Virtex UltraScale+ FPGAs.

The authors demonstrated that this system can sort 230 key-value records 37.4 times faster than GNU parallel sort running on a CPU with 8 threads. However, their method assumed collaboration between the CPU and FPGA, and the sorting performance was ultimately limited by the PCIe bandwidth, which was 7.2 GB/s as reported in [14]. Moreover, their method was not suitable for implementing FPGA-centric applications that require sorting due to its structure.

Shinyamada et al. [38] evaluated the impact of various sorts on high-level synthesized image processing hardware. The results showed that bubble-sort and odd-even-merge-sort were the fastest algorithms, as they were able to achieve pipeline processing. Conversely, selection-sort was not able to achieve ideal pipeline processing, and its performance was not as good as the former two algorithms. Concludingly, optimizing sorts can have a significant impact on overall image processing performance.

Moghaddamfar et al. [32] conducted a comparative analysis of OpenCL and RTL-based implementations of a heap-sort that merges sorted runs. The results showed that while both implementations required comparable development effort, their RTL implementations of critical primitives achieved four times better performance and used only half as much FPGA resources compared to the OpenCL implementation.

This highlighted the importance of carefully selecting the programming language and implementation approach when developing algorithms for FPGA-based systems. The study suggested that RTL-based implementations can provide significant performance benefits and resource efficiency compared to higher-level language implementations like OpenCL.

Chen et al. [15] proposed a novel hybrid pipelined sorting architecture based on a bitonic-sorter and several cascaded sorting units. This sorting architecture achieved a balanced trade-off between resource utilization and throughput, as well as between throughput and power consumption.

Specifically, the architecture was both resource and energy efficient in terms of the throughput-to-resource ratio and the throughput-to-power ratio. This study highlighted the importance of designing sorting architectures that maximize data parallelism to achieve increased throughput and reduced latency.

Montesdeoca et al. [33] monitored by a network of 40 CO_2 sensors and performed real-time sorting of all the data via bubble-sort and insertion-sort on FPGA. The results showed that insertion-sort was faster than bubble-sort, but it consumed more hardware resources in the FPGA, illustrating the importance of the trade-off between speed and resource utilization when selecting a sort for real-time applications.

In [3], the authors evaluated multithreaded sorts on a 32-core reconfigurable architecture with embedded real-time Linux support. The architecture consisted of NIOS II/f soft cores and was implemented on an FPGA. The authors proposed a new approach for performance evaluation of a soft multithreaded multicore architecture conducted in real-time.

This approach was based on the recursive generation and execution of sorts such as merge-sort and quick-sort. The architecture was capable of achieving high parallelism and throughput while maintaining low latency. The architecture outperformed the others in terms of speed and scalability. In [1], the authors focused on hardware implementing bubble-sort, selection-sort, insertion-sort, merge-sort, bitonic-sort and odd-even-merge-sort using FPGA in synchronous and pipelined architectures. The authors compared these implementations in terms of computational time and area.

They showed that non-pipelined bitonic-sort and non-pipelined odd-even-merge-sort had the best performance in terms of computational time, while the non-pipelined selection-sort and non-pipelined insertion-sort had the lowest area of synchronous architecture.

For pipelined architectures, bitonic-sort and odd-even-merge-sort had much lower computational time when implemented in hardware. Additionally, odd-even-merge-sort was found to be the smallest in terms of area. While bitonic-merge-sort was slightly larger in area and slower in execution than odd-even-merge-sort.

Lobo et al. [31] compared five merge-sorts (serial-merge-sort, parallel-merge-sort, bitonic-merge-sort, odd-even-merge-sort and the modified-merge-sort) in terms of resource utilization, delay and area on FPGA. The results showed that the serial and parallel merge use the highest amount of resource utilization compared to bitonic-merge, odd-even-merge and modified-merge.

Also, the parallel-merge algorithm was much faster than a serial-merge algorithm. In addition, the odd-even and modified-merge had a very close value of the area used while bitonic-merge had as slightly higher value.

Abdelrasoul et al. [2] proposed an index and sort algorithm (IaSA) based on an FPGA (vertex-5 series) in a pipelined sequential structure using Verilog HDL. The results showed that, for various data set sizes, IaSA performed best in terms of computational time.

In [24], the authors presented a column-sort, mapped on an HBM (High Bandwidth Memory)-enabled FPGA. The approach utilized computational pipelines, hardware-efficient interconnection networks and several optimizations to achieve high-throughput sorting. The results showed that their optimized design yields 14.8×, 4.73× and 2.18× speedup compared with state-of-the-art implementations on CPU.

– Lines 5 to 8: First step of the algorithm, check for external variations from the computational environment.

– Line 8: badRSDR$[s]$ $\leftarrow$ true if unstable computational environment (let's try to continue anyway).

Table 1. Review of sorting algorithms on different platforms

Approach	Platforms			Algorithms			HLS	Optimization	Application domain
Ref	**FPGA**	**Intel CPU**	**ARM**	**Sort-name**	**Complexity**	**Perf. criteria**			
[26, 27]	Yes	No	No	Library sorting	O(n log(n))	Resources utilization	Yes	yes (openCL)	Sorting data
[9]	Yes	No	No	bubble, selection	O(n2̂)	Resources utilization, computational times, standard deviation	Yes	Yes	ITS
				insertion, quick	O(nˆ2)				
				shell	O(nˆ(3/2))				
				merge,heap,tim	O(n log(n)				
[14]	Yes	No	No	sample	O(n log(n))	Computational times	No	yes	Amazon web server
[38]	Yes	No	No	bubble, selection,	O(nˆ2)	Computational times	Yes	Yes	Image processing
				odd-even-merge					
[32]	Yes	No	No	heap, merge	O(n log(n))	FPGA resource usage, development effort	No	Yes (OpenCL)	Sorting data
[15]	Yes	No	No	bitonic	O(log(n)ˆ2)	FPGA resource usage, energy efficient	No	Yes (Pipeline)	Real World Application
[33]	Yes	No	No	bubble, insertion	O(nˆ2)	FPGA resource usage, computational times	No	Yes	Wireless sensor network on IoT
[3]	Yes	No	No	quick	O(n2̂)	Parallelization efficiency, computational times	RTL	Yes (POSIX thread)	-
				merge	O(n log(n))				
[1]	Yes	Yes	No	bubble, selection	O(n2̂)	Computational times, area	No	Yes	Data processing
				insertion	O(nˆ2)				
				merge	O(n log(n))				
[31]	Yes	No	No	serial merge,	O(n log(n))	Resource utilization, delay, area	No	No	Particular application
				parallel merge,	O(log(n))				
				bitonic-merge,	O(log(n)ˆ2)				
				odd-even-merge	O(log(n)ˆ2)				
				modified merge	O(log(n)ˆ2)				
[2]	Yes	No	No	index		Computational times	No	Yes (pipeline)	-
[24]	Yes	No	No	hyper, column	O(n log(n))	External memory (HBM)	-	Yes	-
[11, 10]	Yes	No	Yes	insertion, quick	O(n2̂)	Computational times, energy consumption, temporal stability	Yes	Yes	Intelligent systems (ITS)
				shell	O(nˆ(3/2))				
				heap, merge, tim	O(n log(n))				
[7]	Yes	No	Yes	heap	O(n log(n))	Power, Speedup	No	Yes	Wavelet Based Image Coder
[8]	Yes	No	Yes	heap	O(n log(n))	Power, Speedup	No	Yes	Image coding
[25]	Yes	No	Yes	heap	O(n log(n))	Energy consumption	No	Yes	Embedded System
[12]	No	No	Yes	network sorting	O(log(n)ˆ2)	Speedup	No	Yes	Commercial microchips
[30]	No	Yes	No	counting,	O(n)	Computational times	No	Yes (parallelism)	Sorting data
				bucket,	O(nˆ2)				
				merge,	O(n log(n))				
[21]	No	Yes	No	bitonic,	O(log(n)2̂)	Computational times, memory	No	Yes (OpenMp)	Sorting data
				merge,	O(n log(n))				
[4]	No	Yes	No	insertion, quick	O(n2̂)	Time, stability, memory space	No	No	Data base, Network, AI
				bubble,selection	O(nˆ2)				
				merge	O(n log(n))				
[28]	No	Yes	No	insertion, stl	O(nˆ2)	Time	No	Yes	Computer science
Our work	Yes	Yes	No	quick	O(n2̂)	Computational times, temporal stability	Yes	Yes (HLS directives)	Intelligent systems (ITS)
				shell	O(nˆ(3/2))				
				heap, merge, tim	O(n log(n))				

Algorithm 1: Main steps of our DOE

```
Input  : A given architecture (i7 or FPGA).
         A list of sorts $\mathcal{S}$.
         A value of $n \in \mathcal{N}$.
         A set $\mathcal{P}_n$ of $P$ permutations $\pi_n$.
Output: A partial or total ranking of the sorts in
         the ordered list $\mathcal{S}'$.
         Temporal stability of sorts in stable[$\cdot$].
         Out of range RSD$_R(s, \pi_n)$ in badRSDR[$\cdot$].
1  $\mathcal{S}' \leftarrow \emptyset$
2  foreach sort $s \in \mathcal{S}$ do
3      stable[$s$] $\leftarrow$ false
4      badRSDR[$s$] $\leftarrow$ false
5      foreach permutation $\pi_n \in \mathcal{P}_n$ do
6          compute RSD$_R(s, \pi_n)$
7          if RSD$_R(s, \pi_n) > 5\%$ then
8              badRSDR[$s$] $\leftarrow$ true
9      compute $\mu_P(s)$, $\sigma_P(s)$, RSD$_P(s)$ and CI$(s)$
10     if RSD$_P(s) \leq 5\%$ and $Q2(s) \approx \mu_P(s)$ then
11         stable[$s$] $\leftarrow$ true
12         $\mathcal{S}' \leftarrow \mathcal{S}' + s$   ▷ rank $s$ according to its $\mu_P$
13     else
14         compute $IQR(s)$
15         if $s$ intersects with other sorts $\in \mathcal{S}'$ then
16             perform statistical tests
17             $\mathcal{S}' \leftarrow \mathcal{S}' + s$       ▷ rank $s$ via p-values
18     compute #$^+$outliers($s$) and %$^+$outliers($s$)
19     adjust stable[$s$] according to %$^+$ outliers($s$) if needed
20     adjust stable[$s$] according to badRSDR[$s$] if needed
```

– Lines 10 to 12: RSD$_P(s)$ is sufficiently low to avoid using sophisticated statistical tests to rank the sorts.

– Line 10: RSD$_P(s) \leq 5\%$ and $Q2(s) \approx \mu_P(s)$ suggest that the sort s is stable and can be ranked via line 12.

– Line 12: RSD$_P(s) \leq 5\%$ is considered as sufficient for $\mu_P(s)$ to be statistically representative.

CI(s) is compared with those of previously ranked sorts and used to assess whether a partial or a total ranking of the sorts is possible when inserting and ranking s in $\mathcal{S}'$.

– Lines 13 to 17: In this step, statistical tests are required due to a high value of RSD$_P(s)$ or a significant difference between $Q2(s)$ and $\mu_P(s)$.

– Line 15: Current confidence interval CI(s) (resp. $IQR(s)$) is compared with those of previously ranked sorts in $\mathcal{S}'$.

– Line 16: Statistical tests are used to rank the sorts if possible, temporal stability tests of s are performed elsewhere.

– Line 17: At this point it is not always possible to strictly rank s. In this case, $\mathcal{S}'$ contains a partial ranking of the sorts.

– Lines 18 to 20: Check if sort s is compatible with worst-case computational times (i.e. upper outliers). With line 10, this last step assesses/adjusts the stability of s in line 19.

– Line 19: Set stable[s] to false if %$^+$outliers$(s) > 5\%$.

– Line 20: Set stable[s] to false if badRSDR[s] $=$ true to take unstable computational environment into account.

2.2 Software Acceleration Methods for Embedded Systems (ARM)

ARM processors offer advantages in terms of flexibility and ease of reconfigurable integration technology for a wide range of applications, from embedded systems to high-performance computing.

Compared to classical processors, soft-core processors like ARM allow for greater customization and adaptability because they can be easily programmed and reconfigured to meet the specific needs of a given application.

Additionally, soft-core processors can be integrated into a variety of reconfigurable technologies, such as FPGAs, allowing for even greater flexibility and performance optimization.

In [11, 10], the authors proposed a software implementation for insertion-sort, quick-sort, heap-sort, shell-sort, merge-sort, and tim-sort on an ARM Cortex A9. They compared the performance of these algorithms in terms of average and standard deviation of computational times, energy consumption, and temporal stability.

Table 2. Average and standard deviation of computational times on i7

size/ns	tim-sort	merge-sort	heap-sort	shell-sort	quick-sort
8	28.3 (2.2)	83.7 (2.2)	53.0 (3.5)	38.2 (1.2)	47.2 (2.7)
16	49.6 (7.7)	164.5 (4.8)	121.3 (5.7)	73.6 (2.9)	87.0 (13.6)
32	127.2 (38.0)	346.1 (14.8)	307.0 (13.8)	168.1 (9.6)	198.0 (72.6)
64	565.4 (167.6)	752.4 (45.8)	682.2 (35.8)	406.7 (24.6)	507.8 (364.8)
128	1285.8 (405.9)	1685.4 (147.6)	1582.5 (76.3)	1031.2 (95.9)	1374.3 (1399.5)
256	3226.4 (1101.5)	3622.4 (316.8)	3954.4 (136.7)	3716.8 (732.3)	4003.8 (5396.8)
512	9222.3 (3192.9)	9374.1 (1355.0)	9833.0 (318.2)	15199.5 (4159.4)	14837.7 (20382.2)
1024	32528.8 (11768.7)	33140.8 (7954.5)	23720.2 (852.1)	40367.4 (11791.8)	53518.4 (80339.7)
2048	84812.9 (31077.6)	82910.9 (21840.3)	56650.7 (2792.7)	100684.9 (30593.3)	182912.0 (329092.6)
4096	189435.6 (69730.3)	183894.0 (49724.5)	129825.6 (7041.9)	235353.3 (72545.8)	641826.5 (1346751.2)

The results demonstrated that shell-sort was the best algorithm, being 42.1% faster and even reaching up to 72% faster when the number of elements to be sorted is greater than 64.

However, when the number of elements is smaller than 64, tim-sort was the best algorithm. Additionally, shell-sort was the best algorithm in terms of standard deviation of computational times and energy consumption.

In [7], the authors proposed a hardware heap-sort implementation using FPGA of a wavelet based image coder. Their architecture provided up to 20.9% power reduction on the memories compared to the baseline implementation. Moreover, their architecture provided 13x speedup compared to ARM Cortex A9.

In [8], the authors introduced an adaptive heap-sort that was designed for an image coding implementation on FPGA with high throughput and scalable sorting. The authors compared its performance to an embedded ARM Cortex A9 running at 666 MHz.

Their architecture, running at 100 MHz, provided around 13 times the speedup while consuming 242 mW of average core dynamic power. In [12], the authors adapted a hybrid sort based on quick-sort and bitonic-sort. They employed bitonic-sort to handle small partitions/arrays with a vectorized partitioning implementation to divide these partitions.

Their approach required only an array of size O(log n) for recursive calls in the partitioning phase. They evaluated the performance on an ARM v8.2 (A64FX) and assessed their implementation by sorting/partitioning integers, double floating-point numbers, and key/value pairs of integers. The results showed an average speedup factor of four compared to the GNU C++ sort algorithm.

In [25], the performance and energy efficiency of hardware and software implementations of the heap-sort are compared. The results showed that the hardware implementation (Digilent Basys 3 Artix-7 FPGA) was more energy efficient, but slower than software implementation (ARM Cortex A72) due to a low clock frequency.

2.3 Software Acceleration Methods for Non Embedded CPU

In “standard” workstations, Central Processing Units (CPUs) can drastically increase the number of instructions processed per second, allowing the computer to perform more complex tasks or run more programs simultaneously. Indeed, recent CPUs have a potential for higher performance thanks to higher clock speeds, more cores, and improved SIMD instructions. However, these processors consume much more power than their embedded counterparts.

In [30], the authors presented optimized serial and parallel counting-sorts. They compared this

Table 3. RSD$_P(s)$ of sorting algorithms on i7

size/%	tim-sort	merge-sort	heap-sort	shell-sort	quick-sort
8	7.8	2.6	6.7	3.2	5.7
16	15.6	2.9	4.7	4.0	15.7
32	29.9	4.3	4.5	5.7	36.7
64	29.6	6.1	5.2	6.1	71.8
128	31.6	8.8	4.8	9.3	101.8
256	34.1	8.7	3.5	19.7	134.8
512	34.6	14.5	3.2	27.4	137.4
1024	36.2	24.0	3.6	29.2	150.1
2048	36.6	26.3	4.9	30.4	179.9
4096	36.8	27.0	5.4	30.8	209.8

sort to others such as bucket-sort and merge-sort, implementing both counting-sort and merge-sort on CPU and GPU. The results showed that the optimized counting-sort took only 6 ms to sort 100 million integers, being 23 times faster than the previous version.

In [21], the authors provided an analysis of current host-GPU data transfer mechanisms and explored methods for mitigating performance bottlenecks. They developed a heterogeneous CPU/GPU sort and demonstrated that while out-of-place GPU sorting achieved the best performance, an in-place sort further reduced some host-side bottlenecks.

In [4], the authors compared the grouping-comparison-sort (GCS) to selection-sort, quick-sort, insertion-sort, merge-sort and bubble-sort, using random input sequences. On an Intel Core 2 Duo E8400 @ 3.00 GHz (2 CPUs), the result revealed that for small input sizes, the performance of all six algorithms was almost comparable.

However, for larger input, quick-sort proved to be the fastest, while selection-sort was the slowest. GCS ranked as the third fastest for small input size (10000 elements) and the fifth fastest for large input size (30000 elements). In [28], the authors introduced a distribution sorting method that utilized a trained model of the empirical Cumulative Distribution Function of the data.

Additionally, they applied a deterministic sort that performed well on almost sorted arrays, such as insertion-sort. The performance was measured on an Intel Xeon Gold 6150 @ 2.70 GHz using up to one billion double-precision keys following a normal distribution.

Their approach achieved an average performance improvement of 3.38 times compared to the C++ STL-sort, which is an optimized hybrid of quick-sort, a 1.49 times improvement over sequential radix-sort, and a 5.54 times improvement over a C++ implementation of tim-sort, which is the default sorting function for Java and Python.

2.4 Synthesis of Related Work

Table 1 summarizes a literature review focusing on the use of several sorts on different platforms (FPGA, Intel CPU, ARM). This summary shows that the authors generally use sorts on FPGA to improve performance in terms of computational time and resource usage. Moreover, a majority of authors do not use HLS except for [26, 27, 38, 9, 11, 10]. In addition, the number of elements to sort is usually much larger than thousands of items. Contrary to these studies, our work is based on different implementations of sorts using HLS and compare the sorts on FPGA and Intel i7 in

Table 4. Ranking of sorts using $\mu_P(s)$ and CI(s) on i7

size					$\mathcal{S}'$				
8	tim	$\prec$	shell	$\prec$	quick	$\prec$	heap	$\prec$	merge
16	tim	$\prec$	shell	$\prec$	quick	$\prec$	heap	$\prec$	merge
32	tim	$\prec$	shell	$\prec$	quick	$\prec$	heap	$\prec$	merge
64	shell	$?$	quick	$?$	tim	$\prec$	heap	$\prec$	merge
128	shell	$\prec$	tim	$?$	quick	$?$	heap	$\prec$	merge
256	tim	$?$	merge	$?$	shell	$?$	heap	$?$	quick
512	tim	$?$	merge	$?$	heap	$?$	quick	$?$	shell
1024	heap	$\prec$	tim	$?$	merge	$\prec$	shell	$?$	quick
2048	heap	$\prec$	merge	$?$	tim	$?$	shell	$?$	quick
4096	heap	$\prec$	merge	$?$	tim	$\prec$	shell	$\prec$	quick

terms of computational time, resources utilization and temporal stability.

It is worth noting that while many authors are interested in the first two criteria, the number of papers concerning the stability of algorithms is much lower. Moreover, since an avionics application is targeted, deterministic sorting is required and the use recursive functions or dynamic memory allocations are forbidden contrary to many applications in the literature.

Similarly, parallel versions of sorts are not allowed because it is not possible to certify such algorithms on standard multicore architectures in our target avionics application.

A finding from this synthesis is that beyond its inherent complexity, the “best sort” depends on the number of elements to be sorted, the target architecture, the parallelization mode and the considered key performance indicators. In the following sections, we compare software and hardware implementations of sorts on Intel i7 and FPGA.

3 Experimental Results

In this study, the performances of five sorts $s \in \mathcal{S}$ are compared on Intel i7 and FPGA architectures: $\mathcal{S} = \{$heap-sort, quick-sort, merge-sort, shell-sort, tim-sort$\}$. The number of sorts corresponds to the cardinality of $\mathcal{S}$ and is denoted by $\Sigma = |\mathcal{S}|$. The sorts are evaluated in terms of computational times and temporal stability.

Considering sorting algorithms, the usual informal definition of the stability is the following: A sort is ideally stable if it maintains the relative order of elements with equal values. This means that whenever there are two elements a and b with the same value, the relative order of a and b is preserved by the sort.

However, this study focuses on temporal stability, defined as follows: A sort is temporally stable if its computational time is independent of the order of the elements to be sorted.

Although target application requires deterministic sorts, there is no guaranty (and no need) of usual stability in our implementations of the sorts since the resulting relative order of a and b (with the same value) will deterministically be the same after the sort but this is not necessarily the initial relative order of a and b (before the sort). Now that we have defined the evaluation criteria, next section describes our design of experiments.

3.1 Design of Experiments

In order to evaluate the performances of sorts, the average and standard-deviation of their computational times are studied using $n = 8$ to 4096 elements $n \in \mathcal{N} = \{$8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096$\}$, via P =47 permutations of n integers generated using Lehmer’s method [17]. For each value of n, this set of permutations $\mathcal{P}_n$ is used to characterize the temporal variations due to a sort s in itself, i.e. its temporal stability.

For each permutation π_n of size n, $R = 1000$ replications are used to identify the external variations coming from the “computational environment” (e.g., transmission error, operating system noise, I/O buffering). To further evaluate the stability of sorts, the Relative Standard Deviation (RSD) is calculated by dividing the standard deviation σ by the average of computational times μ and expressed as a percentage:

$$\text{Relative\_Standard\_Deviation} = 100 \times \frac{\sigma}{\mu}. \quad (1)$$

Table 5. Ranking of sorts using statistical tests on i7

size					$\mathcal{S}'$				
8	tim	$\prec$	shell	$\prec$	quick	$\prec$	heap	$\prec$	merge
16	tim	$\prec$	shell	$\prec$	quick	$\prec$	heap	$\prec$	merge
32	tim	$\prec$	shell	$\preceq$	quick	$\prec$	heap	$\prec$	merge
64	quick	$\prec$	shell	$\prec$	tim	$\prec$	heap	$\prec$	merge
128	quick	$\prec$	shell	$\prec$	tim	$\prec$	heap	$\prec$	merge
256	quick	$\prec$	tim	$\prec$	merge	$\prec$	shell	$\preceq$	heap
512	quick	$\prec$	merge	$\preceq$	heap	$\prec$	tim	$\prec$	shell
1024	heap	$\preceq$	quick	$\prec$	merge	$\prec$	tim	$\prec$	shell
2048	heap	$\prec$	quick	$\prec$	merge	$\prec$	tim	$\prec$	shell
4096	heap	$\prec$	quick	$\prec$	merge	$\prec$	tim	$\prec$	shell

It is a relative measure of the dispersion of data around the average. Ultimately, this ratio is used to compare the degree of variation from one sample to another, even if the means are different. For a sufficiently large number of items in a population (empirically $\geq$ 30), an RSD of less than 1% is considered “excellent” to make the average representative.

From a practical point of view, an RSD of 5% is generally considered acceptable. However, if RSD is greater than 5%, it is advisable to use statistical tests, possibly supplemented by graphical representations such as boxplots. Our study relies on two measures of RSD:

For each permutation π_n of size n, RSD$_R(s, \pi_n)$ is computed over the set of R replications of the same computational time measurement (i.e. for a given sort s, sorting the same permutation π_n).

Then, for each sort s, RSD$_P(s)$ is computed over the set $\mathcal{P}_n$ of P permutations, considering – for each permutation π_n – the average computational time $\mu_R(s, \pi_n)$ over R replications of the same computational time measurement. To reduce RSD$_R(s, \pi_n)$, the operating system (Debian Linux 12.4 with “processor affinity for RT-tasks” based on kernel version 6.1.0-17) has been configured to avoid most of the OS’s noises, a disruption may however occur due to potential non-maskable interruptions and unavoidable waiting times for external events or resource availability.

The values of RSD$_R(s,\ \pi_n)$ are not detailed for the sake of conciseness. R has been chosen so that RSD$_R(s,\ \pi_n)\ \leq\ 5\%$ in “almost all cases” (i.e. for the large majority of the sorts and values of n). However, due to above-mentioned external variations, this target is not reached for all sorts with $n = 8$ on i7 and FPGA as well as for quick-sort and tim-sort with $n = 16$ on i7.

Nonetheless, it should be noted that the average RSD$_R(s, \pi_n)$ over all sorts s and all permutations π_n is approximately equal to 1.2% which is far less than the expected limit of 5%. In our design of experiments, RSD$_P(s)$ is computed for each sort s to assess the representativeness of its average computational time.

Actually, the stability of sorts in terms of computational time is assumed significant if RSD$_P(s)\ \leq\ 5\%$, otherwise statistical tests are performed. Moreover, in order to rank the sorts, confidence intervals CI(s) are computed as follows:

$$\text{CI}(s) = \left[\mu_P(s) - 1.96 \cdot \frac{\sigma_P(s)}{\sqrt{P}}, \mu_P(s) + 1.96 \cdot \frac{\sigma_P(s)}{\sqrt{P}}\right]. \quad (2)$$

In addition to the focus on a single sort, boxplots [41] are useful tools for visualizing and comparing distributions of computational time measurements on the same scale. Specifically, boxplots allow us to compare and analyze the stability of different algorithms.

Standard boxplots are usually based on five values that summarize the data ($Q0$ (min), $Q1$ (first quartile), $Q2$ (median), $Q3$ (third quartile), and $Q4$ (max)) for the studied population (here, 47 permutations). As usual, IQR is also defined as the difference between the third and first quartiles. All observations above $Q3 + 1.5 \times IQR$ or below $Q1 - 1.5 \times IQR$ are considered as outliers.

On the basis of the number of outliers # outliers, this allows us to compute the percentage of outliers as follows:

$$\text{\# outliers}(s) = 100 \times \frac{\text{\#outliers}(s)}{P}. \quad (3)$$

Upper outliers (denoted by #$^+$) are defined as all observations above $Q3 + 1.5 \times IQR$. Similarly, %$^+$outliers is the percentage of upper outliers. Each set of P experiments (on P distinct

Table 6. Upper outliers percentages $\%^{+}$ outliers on i7

size	tim-sort	merge-sort	heap-sort	shell-sort	quick-sort
8	2.13	0.00	0.00	0.00	6.38
16	4.26	8.51	0.00	0.00	10.64
32	8.51	2.13	2.13	8.51	12.77
64	0.00	0.00	6.38	4.26	12.77
128	6.38	0.00	6.38	8.51	12.77
256	6.38	0.00	6.38	2.13	12.77
512	0.00	0.00	0.00	6.38	12.77
1024	2.13	0.00	0.00	4.26	12.77
2048	2.13	0.00	0.00	0.00	12.77
4096	0.00	0.00	0.00	4.26	12.77

permutations) of a sort s may be represented by a boxplot and summarized by $Q0(s)$ to $Q4(s)$.

Supplemented by the outliers, $\sigma_P(s)$, $\mu_P(s)$ and RSD$_P(s)$, it is possible to obtain a precise view of the temporal stability of a sort. RSD$_P(s) < 5\%$ and/or a small value of $IQR(s)$ suggest(s) that the sort s is stable, this is usually confirmed by a small value of %outliers(s).

On the contrary, RSD$_P(s) > 5\%$ and/or a large value of $IQR(s)$ and/or a significant difference between $Q2(s)$ and $\mu_P(s)$ suggest(s) that the sort is not stable. In the context of real-time applications, particular attention should be payed to the percentage of upper outliers ($\%^{+}$outliers) since it provides information on worst-case computational times of a given sort.

In other words, depending on input data this sort may reach computational times that are not compatible with the targeted time-constraints.

Moreover, by grouping several boxplots (one per sort) in the same plot, it is possible to visually compare the performance and temporal stability of the sorts. In addition to $\mu_P(s)$ and RSD$_P(s)$, a significant variation of $Q2(s)$ between different distributions (i.e. sets of P permutations, one set per sort s), suggests that the sorting times vary and it is possible to rank the sorts.

In contrast, if the medians are quite similar across several distributions, $\sigma_P(s)$, $\mu_P(s)$ and/or boxplots are not sufficient to rank the sorts and statistical tests are required. Consequently, it should have been interesting to present boxplots and statistical tests for all sorts on all target architectures, however for the sake of conciseness, numerical results have been reduced to $\mu_P(s)$, $\sigma_P(s)$, RSD$_P(s)$ and %outliers(s) when/where sufficient. The results of statistical tests are also summarized when needed.

In our design of experiments, the following statistical tests are performed: First of all, to assess the normality of the data distributions, the Shapiro-Wilk test is conducted on each set of P experiments and each value of n. As a result of these tests, it appears that the data deviate significantly from a normal distribution, consequently nonparametric tests must be used.

So, in a second step, the Kruskal-Wallis test is employed to examine the overall differences in computational times among the five sorts on each platform. Subsequently, to refine the results and rank the sorts as far as statistically possible, pairwise comparisons using Wilcoxon tests are conducted to identify specific algorithm pairs that exhibited significant differences.

For a given architecture, if all tests are performed on the $\eta = |\mathcal{N}|$ values of n, this leads to $\eta \times \Sigma$ Shapiro-Wilk tests, η Kruskal-Wallis tests and $\eta \times (\Sigma \times (\Sigma - 1))/2$ Wilcoxon tests. This makes a total of – at most – 160 statistical tests (with $\eta = 10$

Table 7. Average and standard deviation of computational times on FPGA

size/us	tim-sort	merge-sort	heap-sort	shell-sort	quick-sort
8	18.12 (3.47)	18.93 (3.49)	17.99 (3.49)	17.88 (3.48)	18.68 (3.51)
16	22.07 (3.47)	23.60 (3.48)	21.85 (3.48)	22.25 (3.48)	23.22 (3.49)
32	31.18 (3.47)	34.48 (3.47)	31.10 (3.48)	33.25 (3.48)	34.30 (3.48)
64	50.89 (3.47)	58.46 (3.48)	52.78 (3.47)	60.24 (3.48)	62.80 (3.48)
128	94.50 (3.47)	111.88 (3.49)	102.67 (3.49)	126.30 (3.48)	137.90 (3.48)
256	189.00 (3.47)	231.30 (3.48)	228.57 (3.48)	293.55 (3.48)	351.20 (3.48)
512	393.17 (3.47)	482.50 (3.49)	466.05 (3.48)	668.20 (3.48)	1001.30 (3.48)
1024	832.85 (3.47)	1031.50 (3.48)	1015.50 (3.47)	1625.10 (3.56)	3121.00 (3.49)
2048	1769.50 (3.47)	2211.50 (3.47)	2227.10 (3.49)	4027.67 (3.48)	10660.00 (3.48)
4096	3756.00 (3.47)	4734.90 (3.48)	4848.50 (3.48)	9756.40 (3.48)	38507.00 (3.49)

and $\Sigma = 5$) performed via several scripts written in R language [23].

The effective number of pairwise comparisons can be reduced if it is guided by the ranking of sorts by μ_P. All p-values are adjusted using the Bonferroni correction method so as to obtain an overall α level set to 5%.

For a given architecture (i7 or FPGA) and a value of n, our DOE follows several steps described in a simplified way as a pseudo-code in Algorithm 1, supplemented by some comments about main lines of the algorithm.

As shown in previous paragraphs, some steps of Algorithm 1 are dedicated to check the stability of the computational environment while others are used to assess the temporal stability of each sort and the others deal with the total or partial ranking of the sorts on the basis of their computational times. Finally, it is important to note that the resulting ranking in $\mathcal{S}'$ is not necessarily the same when n varies.

3.2 Performances Study of Sorting Algorithms on i7

This section illustrates our DOE to compare and analyze the sorts on an Intel i7-9850H @ 2.60 GHz. All codes are written in C language and compiled via gcc version 12.2.0-14 (on Linux Debian 12.4) with the O3 optimization flag turned on. The sorts are evaluated in terms of computational times. They are then rated on the basis of their temporal stability.

In more details, the first step of our DOE computes RSD$_R(s)$ to check for external variations from the computational environment (lines 5 to 8 of Algorithm 1). Then, subsection 3.2.1 illustrates the second step of Algorithm 1 (lines 10 to 12). In a similar way, subsection 3.2.2 follows the third step of Algorithm 1 (lines 13 to 17). Finally, subsection 3.2.3 is dedicated on the last step of Algorithm 1 (lines 18 to 20).

We detail each of the steps 2 to 4 of Algorithm 1 in a separate subsection to illustrate how it works and “its” results in several tables, but in reality the steps follow one another in the algorithm to lead if necessary to statistical tests and temporal stability analysis.

3.2.1 Average and Standard-Deviation of Computational Times on i7

To illustrate the second step of Algorithm 1 (lines 10 to 12) Table 2 displays the average and

Table 8. RSD$_P(s)$ of sorting algorithms on FPGA

size%	tim-sort	merge-sort	heap-sort	shell-sort	quick-sort
8	19.15	18.44	19.40	19.43	18.79
16	15.72	14.75	15.91	15.61	15.03
32	11.13	10.07	11.10	10.46	10.14
64	6.82	5.94	6.57	5.76	5.54
128	3.67	3.10	3.38	2.75	2.52
256	1.83	1.50	1.51	1.18	0.99
512	0.88	0.44	0.78	0.52	0.34
1024	0.42	0.34	0.40	0.22	0.11
2048	0.20	0.16	0.16	0.09	0.03
4096	0.09	0.07	0.07	0.04	0.00

standard deviation of computational times ($\mu_P(s)$ and $\sigma_P(s)$) whereas Table 3 shows the relative standard deviation RSD$_P(s)$. In this subsection, the objective is to rank the sorts according to $\mu_P(s)$ while considering RSD$_P(s)$ and CI(s) as in the second step of Algorithm 1.

In this step, RSD$_P(s)$ and CI(s) are not used to assess the temporal stability, but respectively to check whether statistical tests are needed or whether it is possible to obtain a total or partial ranking of the sorts. On i7, these measurements lead to the ranking of the sorts presented in Table 4.

In this table, the sorts are ranked from left (first/best/fastest sort) to right (last/worst/slowest sort) according to their $\mu_P(s)$. The $\prec$ symbol indicates a strict ranking, with no intersection of confidence intervals whereas $?$ denotes that the confidence intervals overlap.

At this second step of Algorithm 1, based on $\mu_P(s)$ and CI(s), Table 2 and Table 4 show that for $n < 64$, tim-sort outperforms other sorts in terms of μ_P. Tim-sort, however, has a higher RSD$_P(s)$ than merge-sort, heap-sort, and shell-sort, which is not a good clue – at this step – of its temporal stability.

Similarly, for $n > 512$, heap-sort outperforms other sorts in terms of μ_P. In between, the rankings are not clearly established due to intersections of the confidence intervals and it is challenging to draw a significant conclusion regarding the best algorithm.

For $n \geq 16$, it should be noted in Table 3 that RSD$_P(s)$ for heap-sort is less than 5% (or slightly greater than 5%) and $Q2(S) \approx \mu_P(s)$ (the maximum gap between $Q2(S)$ and $\mu_P(s)$ is equal to 2%) leading Algorithm 1 to line 12. However, since other sorts have higher values of RSD$_P(s)$ and a gap between $Q2(S)$ and $\mu_P(s)$ greater than 5%, statistical tests will be used for pairwise comparisons in the next step.

As a detail of step 2 of Algorithm 1, we first present a ranking based on the average and the standard deviation of computational times because this is what is conventionally used in the literature, warning however about the intersections between CI(s) that do not allow to establish a total ranking of sorts.

At this step, the results of quick-sort should be viewed with great caution because its RSD$_P(s)$ is really high and there is a very large gap between $Q2(s)$ and $\mu_P(s)$. As for the other sorts, since the RSD$_P(s)$ values are in large majority greater than 5%, statistical tests are used in a next step to confirm or refute the rankings.

With the previous conclusions in mind, we can calculate the relative gains in terms of $\mu_P(s)$, even if the values of RSD$_P(s)$ and CI(s) temper these results. For $n < 64$, tim-sort is 63-70% faster than merge-sort, 46-59% faster than heap-sort,

ISSN 2007-9737

Table 9. Ranking of sorts using $\mu_P(s)$ and CI(s) on FPGA

size					$\mathcal{S}'$					**RSD**$_P$
8	shell	$?$	heap	$?$	tim	$?$	quick	$?$	merge	$> 5\%$
16	heap	$?$	tim	$?$	shell	$?$	quick	$?$	merge	$> 5\%$
32	heap	$?$	tim	$\preceq$	shell	$?$	quick	$?$	merge	$> 5\%$
64	tim	$?$	heap	$\prec$	merge	$?$	shell	$\preceq$	quick	$> 5\%$
128	tim	$\prec$	heap	$\prec$	merge	$\prec$	shell	$\prec$	quick	$\leq 5\%$
256	tim	$\prec$	heap	$\preceq$	merge	$\prec$	shell	$\prec$	quick	$\leq 5\%$
512	tim	$\prec$	heap	$\prec$	merge	$\prec$	shell	$\prec$	quick	$\leq 5\%$
1024	tim	$\prec$	heap	$\prec$	merge	$\prec$	shell	$\prec$	quick	$\leq 5\%$
2048	tim	$\prec$	merge	$\prec$	heap	$\prec$	shell	$\prec$	quick	$\leq 5\%$
4096	tim	$\prec$	merge	$\prec$	heap	$\prec$	shell	$\prec$	quick	$\leq 5\%$

24-32% faster than shell-sort and 36-43% faster than quick-sort.

Furthermore, for $n = 64$ and $n = 128$, shell-sort is 39-46% faster than merge-sort, 35-40% faster than heap-sort, 20-28% faster than tim-sort, and 20-25% faster than quick-sort.

Finally, if $n > 1024$, heap-sort is 56-80% faster than quick-sort, 41-45% faster than shell-sort, 28-32% faster than merge-sort, and 27-33% faster than tim-sort, as mentioned in Table 2. As a conclusion of this step, the rankings of the sorts – at least for $n \geq 64$ – need validations through statistical tests in next step.

3.2.2 Statistical-Tests-Based Ranking on i7

This subsection is based on the third step of Algorithm 1 (lines 13 to 17) which leads to the ranking presented in Table 5. This step is based on nonparametric statistical tests that base their ranking on medians (i.e. $Q2(s)$) and not on averages (i.e. $\mu_P(s)$), unlike the ranking presented in Table 4. This can lead to different rankings if there is a large gap between $Q2(s)$ and $\mu_P(s)$.

This is the case, for example, with the quick-sort, which is ranked first by statistical tests while its results in terms of RSD$_P(s)$ are disastrous as shown in Table 3, relating to previous step of Algorithm 1.

In fact, the quick-sort is tagged as highly unstable by the Algorithm 1 and excluded from the ranking (since for $n > 8$, RSD$_P(s) \gg 5\%$ and there is a very large gap between $Q2(s)$ and $\mu_P(s)$). This explains the use of RSD$_P(s)$ as early as step 2 of the algorithm.

When considering tim-sort, merge-sort and shell-sort, Table 3, shows that their RSD$_P(s)$ is also greater than 5% but the gap between $Q2(s)$ and $\mu_P(s)$ is less that 12% (i.e. far less than the gap for quick-sort) so these sorts are included in current and next step of Algorithm 1.

The results of statistical tests summarized in Table 5 confirm the rankings obtained in Table 4 for $n = 8$, $n = 16$ and $n = 32$. When discarding quick-sort from the rankings, statistical tests confirm the rankings obtained in Table 4 for $n = 64$, $n = 128$, $n = 256$, $n = 2048$ and $n = 4096$ (i.e. respectively shell $\prec$ tim $\prec$ heap $\prec$ merge, shell $\prec$ tim $\prec$ heap $\prec$ merge, tim $\prec$ merge $\prec$ shell $\preceq$ heap, heap $\prec$ merge $\prec$ tim $\prec$ shell and heap $\prec$ merge $\prec$ tim $\prec$ shell). However, for $n = 512$ and $n = 1024$ the rankings are different when comparing Table 5 to Table 4.

Nevertheless, heap-sort is ranked first for $n = 1024$ in the two tables and lead to a total pairwise ranking (denoted by $\prec$ in Table 5). On the contrary, the rankings are different for $n = 512$ in these two tables. It is interesting to note that this ranking, in Table 4, is tagged as not significant since there are intersections of CI(s) (denoted by $?$ in this table).

As a first conclusion of this step, it should be mentioned that the first rankings of the sorts using $\mu_P(s)$ and CI(s) in Table 4 provide us with very accurate rankings for almost all values of n, it has been able to identify the fastest sort in 9 out of 10 cases while discarding the worst one (i.e. quick-sort).

As a second conclusion, this current test, based on statistical test confirms and refines all rankings obtained in previous step, leading to the following choices: Tim-sort is the fastest sort for $8 \leq n \leq 32$ and $n = 256$, shell-sort is the faster sort for $n = 64$ and $n = 128$, heap-sort is the fastest sort for $n \geq 1024$, it is not more than 95% sure that merge-sort is faster than heap-sort for $n = 512$ in this latter case their temporal stabililies is used in next step to refine the choice for $n = 512$.

3.2.3 Temporal Stability of Sorting Algorithms on i7

This subsection is based on the last step of Algorithm 1 (lines 18 to 20) which leads to the results presented in Table 6. First of all, it's worth mentioning that none of these sorts are ideally stable and there are "acceptable" temporal variations. These variations can be bounded by upper values using the upper bound of CI(s) or Q3(s). In this paper, this bound is set to $Q3(s)+1.5\times IQR(s)$, which is the standard definition of upper outliers.

Consequently – at this step of algorithm Algorithm 1 (line 19) – each sort s such as %$^+$outliers$(s) > 0$ should be considered as non temporally stable and discarded from the final ranking of the sorts. It is also important to bound the relative dispersion of computational times around the average and this is exactly the definition of RSD$_P(s)$.

Consequently, each sort s such as RSD$_P(s) > 5\%$ should be considered as non temporally stable and discarded from the while ranking the sorts, either in this current step or in previous steps of Algorithm 1 (line 12 or line 17). Finally, if there are too many variations from the computational environment (Algorithm 1, line 20), it is possible to discard each sort s such that badRSDR$[s] =$ true.

As previously mentioned, it should have been interesting to represent the results using boxplots, however due to lack of space the comparisons are exclusively based on numerical values RSD$_R(s, \pi_n)$, CI(s), RSD$_P(s)$ and %$^+$outliers(s). To this end, RSD$_P(s)$ are given in Table 3 and (%$^+$outliers) are given in Table 6.

Before finalizing the ranking, it is important to mention that in our target application, a duration of less than one microsecond is considered negligible, consequently there no real challenge on temporal stability when considering the sorts for $n \leq 64$ (excepted for quick-sort as previously mentioned).

So for $n \leq 64$ any sort s might be considered as temporally stable, even if %$^+$outliers$(s) > 0$ since its overall duration, including the upper bound of CI(s) or Q3(s) are within the tolerance ranges of one microsecond. As a general comment, it is noticeable that, in Table 3, RSD$_P(s)$ are greater than 1% and even mainly greater than 5%.

Consequently, a good choice might be the fastest sort or the most temporally stable one, that is either tim-sort for $8 \leq n \leq 32$ and shell-sort for $n = 64$ or (fastest sorts) heap-sort for $8 \leq n \leq 32$ and tim-sort for $n = 64$ (most temporally stable sorts) or another combination of the sorts. For $64 \leq n \leq 256$, the best sort in terms of %$^+$outliers(s) is merge-sort whereas the best sort in terms of RSD$_P(s)$ is heap-sort (even if RSD$_P(s)$ for merge-sort are very close to those of heap-sort).

In the same time, for $64 \leq n \leq 256$, the fastest sorts are shell-sort and team-sort. For $n \geq 512$, the best sorts in terms of %$^+$outliers(s) are merge-sort and heap-sort whereas terms of RSD$_P(s)$ is clearly heap-sort. In the same time, for $n \geq 512$, the fastest sorts are also merge-sort and heap-sort. Consequently the best choice for $n \geq 512$ is the heap-sort.

At the end of all the steps of Algorithm 1 we can derive a definitive conclusion regarding the sorting method(s) to be employed for our avionics application on I7. It is clear that the "best sort" on i7 depends on n but also on the criterion to be minimized, i.e. either the computational time or the temporal stability. It is clear that the best choice considering both criteria for $n \geq 512$ is the heap-sort. For $64 \leq n \leq 256$, the choice is not so easy whereas for $n \leq 64$ the computational times and the relating variations are so small that any sort is acceptable on i7.

3.3 Performances Study of Sorting Algorithms on FPGA

As for i7 in section 3.2, this section illustrates our DOE to compare and analyze the sorts on FPGA. All codes are written in C language and the optimized hardware implementation is generated using HLS directives (loop unrolling, loop pipelining, input/output interface). Vivado is used for synthesis and running the VHDL architecture. The sorts are evaluated in terms of computational times and temporal stability.

On FPGA our DOE follows the steps of Algorithm 1: The first step of our DOE computes RSD$_R(s)$ to check for external variations from the computational environment (lines 5 to 8 of Algorithm 1). Then, subsection 3.3.1 illustrates the second step of Algorithm 1 (lines 10 to 12).

Contrary to i7, the values of RSD$_P(s)$ on FPGA are such that there is no need to use statistical tests and the third step of Algorithm 1 is “skipped”. Finally, subsection 3.3.2 is dedicated to the last step of Algorithm 1 (lines 18 to 20). In the following subsections, steps 2 and 4 of Algorithm 1 are detailed to illustrate how it works and “its” results in several tables.

3.3.1 Average and Standard Deviation of Computational Times on FPGA

To illustrate the second step of Algorithm 1, Table 7 displays the average and standard deviation of computational times ($\mu_P(s)$ and $\sigma_P(s)$) whereas Table 8 shows the relative standard deviation RSD$_P(s)$.

In this subsection, the objective is to rank the sorts according to $\mu_P(s)$ while considering RSD$_P(s)$ and CI(s) as in the second step of Algorithm 1. In this step, RSD$_P(s)$ and CI(s) are used to check whether statistical tests are needed or whether it is possible to obtain a total or partial ranking of the sorts. On FPGA, these measurements lead to the ranking of the sorts presented in Table 9.

In this table, the sorts are ranked from left (first/best/fastest sort) to right (last/worst/slowest sort) according to their $\mu_P(s)$. The $\prec$ symbol indicates a strict ranking, with no intersection of confidence intervals whereas $?$ denotes that the confidence intervals overlap. In addition, $\preceq$ indicates that the confidence intervals are contiguous within a range of 1 us, meaning they are “nearly disjoint”.

At this second step of Algorithm 1, based on $\mu_P(s)$ and CI(s), Table 7 and Table 9 show that – in terms of μ_P – for $n = 8$ shell-sort outperforms other sorts, for $n = 16$ or $n = 32$ heap-sort outperforms other sorts, and for $n = 64$ tim-sort outperforms other sorts. However in previous rankings, these “bests sorts” are tagged as not significant since there are intersections of CI(s) (denoted by $?$ in Table 9).

It is worth noting that for $n \leq 64$, RSD$_P(s) > 5\%$ for all sorts, as summarized in Table 9, last column. Before finalizing the ranking, it is important to mention that in our FPGA, $\sigma_P(s)$ is mainly due to hardware perturbations and appears as “almost constant” in Table 7 for all n.

This explains that RSD$_P(s)$ is decreasing as a function of n. This also means that hardware perturbations are too high to rank the sort for $n \leq 64$, which is confirmed by RSD$_P(s) > 5\%$ for all sorts. Therefore, these rankings should be viewed with caution and the hardware perturbations are such that statistical tests would not refine the results.

In Table 9, the results are completely different for $n > 64$ and for all sorts since all pairwise comparisons of sorts are such that confidence intervals of sorts do not intersect (excepted for heap-sort and merge-sort for $n = 256$ where confidence intervals are contiguous within a range of 1 us).

Moreover, for $n > 64$ tim-sort is ranked first and this is confirmed by the fact that RSD$_P(s) < 5\%$ for all sorts, with no need for statistical tests (since $Q2(S) \approx \mu_P(s)$ is also verified), leading Algorithm 1 to line 12. With the previous conclusions in mind, we can calculate the relative gains in terms of

$\mu_P(s)$, even if the values of RSD$_P(s)$ and CI(s) temper these results.

For $n < 64$, heap-sort is 1.01x-1.12x faster than other algorithms. Furthermore, for $n = 4096$, the results show that tim-sort has an average computational time of 3756.00 us, while merge-sort, heap-sort, shell-sort, and quick-sort have average computational times of 4734.90 us, 4848.50 us, 9756.40 us, and 38507.00 us respectively. When comparing tim-sort with the other sorts, the results show that tim-sort is 1.16x-1.21x, 1.08x-1.23x, and 1.25x-1.61x faster than merge-sort, heap-sort, and shell-sort respectively, if $n > 64$.

As a conclusion of this step, the rankings of the sorts on the basis of computational times on FPGA for $n \leq 64$ need to be considered with caution due to the hardware perturbations. Indeed, $\sigma_P(s)$ is almost constant for all sort s and all n, which is due to the synchronous design of the FPGA. Therefore, the average temporal variation does not depend on n (with $n \leq 4096$).

Moreover, Table 9 shows that for $n > 64$, tim-sort is the fastest sort in terms of average computational time. Additionally, there is no intersection of confidence intervals while considering pairwise comparisons of the sorts.

3.3.2 Temporal Stability of Sorting Algorithms on FPGA

This subsection is based on the last step of Algorithm 1 (lines 18 to 20). First of all, it's worth mentioning that none of these sorts are ideally stable and there are "acceptable" temporal variations if $n \leq 64$.

Consequently, each sort s such as RSD$_P(s) > 5\%$ should be considered as non temporally stable and discarded from the while ranking the sorts, either in this current step or in previous steps of Algorithm 1.

Finally, if there are too many variations from the computational environment (Algorithm 1, line 20), it is possible to discard each sort s such that badRSDR$[s]$ $=$ true. Due to the synchronous design of the FPGA and the above-mentioned hardware perturbations, the results of this step can be summarized in a few words.

For $n \leq 64$, RSD$_P(s) > 5\%$ and it is not possible to precisely measure the intrinsic temporal stability of the sorts. For $n > 64$, RSD$_P(s) < 5\%$ (as shown in Table 8) and CI(s) are such that the sorts are considered as temporally stable on FPGA with a constant maximum value for $\sigma_P(s)$ and RSD$_R(s, \pi_n) < 5\%$.

At the end of all the steps of Algorithm 1 we can derive a definitive conclusion regarding the sorting method(s) to be employed for our avionics application on FPGA. It is clear that the "best sort" on FPGA depends on n but the temporal stability is induced by the synchronous design of the FPGA.

This explains that for $n \leq 64$ it is almost impossible to distinguish the sorts from the point of view of computational times as well as from their temporal stability. This also explains the fact that it is possible to clearly distinguish (i.e. with small temporal variations, measured by $\sigma_P(s)$) the "best sort" from the point of view of computational times for $n > 64$, i.e. tim-sort. Moreover, for $n > 64$, the sorts are also undistinguishable from the point of view of their temporal stability.

3.4 Comparison of the Computational Platforms

Despite a much lower frequency, computational times on FPGA are "respectable" compared to those on i7 because Xilink's tool is able to extract the parallelism of the algorithms by means of the optimizations introduced via HLS directives. However, a decrease in terms of average and standard deviation of computational times leads to an increase in resource utilization on FPGA.

Additionally, it is worth noting that the temporal stability of the hardware implementation on FPGA is much better than that on i7, when considering the relative variations given by RSD$_P(s)$ or when considering CI(s).

For conclusion, FPGA provides a better temporal stability than Intel i7 but sorts on FPGA are slower than on i7, even if FPGA offers high performance in terms of parallelism. On the contrary, i7 leads to worse performance than FPGA in terms of temporal stability, when considering RSD$_P(s)$.

However, the sorts run much faster on i7 than on FPGA and it is possible to act on worst-case computational times and limit the time variations by reducing the upper outliers (#$^+$outliers(s)) thanks to dedicated configuration of the operating system, based on "processor affinity for RT-tasks".

The choice of the "best sort" from the points of view of computational time and temporal stability is clear on FPGA and tim-sort appears to be the ranked first. The same choice on i7 is not so easy and depends on n but also on the criterion to be minimized, i.e. either the computational time or the temporal stability. It appears that the best choice considering both criteria for $n \geq 512$ is the heap-sort.

For $64 \leq n \leq 256$, the choice is not so easy whereas for $n \leq 64$ the computational times and the relating variations are so small that any sort is acceptable on i7 when considering the maximum time-constraints of our target application.

4 Conclusions

In this paper, we presented a review of different works using sorting algorithms on Intel i7 and FPGA architectures. To conduct our study, a high-level description of the sorting algorithms is used on FPGA. Our evaluation of various sorts provides valuable insights into their performance and stability, which can guide the selection of suitable algorithms for real-time decision support applications in the avionics industry.

Indeed, a stable sort is particularly useful for real-time targeted applications. In the context of real-time applications, particular attention should be payed to the percentage of upper outliers since it provides information on worst-case computational times of a given sort. In other words, depending on input data this sort may reach computational times that are not compatible with the targeted time-constraints.

The obtained results show that it is difficult to choose the best algorithms on Intel i7, on the contrary tim-sort have a better performance on FPGA for $n \geq 64$. We concluded that the FPGA provides a better performance in terms of temporal stability. We show experimentally that the same sorting algorithms are not ranked in the same way on two different architectures.

Additionally, the calculation of the average and standard deviation of computational times may not be sufficient – depending on the target architecture – to rank these sorts in a statistically representative manner. Similarly, the stability of sorting algorithms may vary from one architecture to another one and/or depending on the size of the data to be sorted. Consequently, we are working on combinations of sorting algorithms to propose a hybrid sort that offers the best possible performance both in terms of computation time and temporal stability.

As future work, we plan to use the hardware version of "the best sorting algorithm" in our targeted avionics decision support system [35, 40, 36]. The present work is also inspired by other researches dedicated to the optimization of matching and scheduling on heterogeneous CPU/FPGA architectures [40] where efficient sorts are required.

References

1. **Abdelrasoul, M., Shaban, A. S., Abdel-Kader, H. (2021).** Based hardware accelerator for sorting data. Proceedings of the 9th International Japan-Africa Conference on Electronics, Communications, and Computations, pp. 57–60. DOI: 10.1109/JAC-ECC54461.2021.9691432.

2. **Abdelrasoul, M., Shaban, A. S., Abdel-Kader, H. (2021).** Index and sort algorithm based on fpga for sorting data. Proceedings of the 9th International Japan-Africa Conference on Electronics, Communications, and Computations, pp. 61–64. DOI: 10.1109/JAC-ECC54461.2021.9691445.

3. **Adam, G. K. (2022).** Co-design of multicore hardware and multithreaded software for thread performance assessment on an FPGA. Computers, Vol. 11, No. 5, pp. 76. DOI: 10.3390/computers11050076.

4. **Al-Kharabsheh, K. S., Al-Turani, I. M., Al-Turani, A. M. I., Zanoon, N. I. (2013).** Review on sorting algorithms a comparative study. International Journal of Computer Science and Security, Vol. 7, No. 3, pp. 120–126.
5. **Almomany, A., Ayyad, W. R., Jarrah, A. (2022).** Optimized implementation of an improved KNN classification algorithm using intel FPGA platform: COVID-19 case study. Journal of King Saud University - Computer and Information Sciences, Vol. 34, No. 6, pp. 3815–3827. DOI: 10.1016/j.jksuci.2022.04.006.
6. **Auger, N., Jugé, V., Nicaud, C., Pivoteau, C. (2020).** Analysis of timsort algorithm. Laboratoire D'Informatique Gaspard-Monge.
7. **Bai, Y., Ahmed, S. Z., Granado, B. (2014).** Fast and power efficient heapsort IP for image compression application. Proceedings of the IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 237–237. DOI: 10.1109/FCCM.2014.72.
8. **Bai, Y., Ahmed, S. Z., Granado, B. (2014).** A power-efficient adaptive heapsort for fpga-based image coding application (abstract only). Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 247. DOI: 10.1145/2554688.2554746.
9. **Ben-Jmaa, Y., Ben-Atitallah, R., Duvivier, D., Ben-Jemaa, M. (2019).** A comparative study of sorting algorithms with FPGA acceleration by high level synthesis. Computación y Sistemas, Vol. 23, No. 1, pp. 213–230. DOI: 10.13053/cys-23-1-2999.
10. **Ben-Jmaa, Y., Duvivier, D., Abid, M. (2021).** Sorting algorithms on ARM cortex A9 processor. Lecture Notes in Networks and Systems, pp. 355–366. DOI: 10.1007/978-3-030-75078-7_36.
11. **Ben-Jmaa, Y., Duvivier, D., Abid, M. (2022).** ARM vs FPGA: Comparative analysis of sorting algorithms. Lecture Notes in Networks and Systems, pp. 275–287. DOI: 10.1007/978-3-030-99619-2_27.
12. **Bramas, B. (2021).** A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE). PeerJ Computer Science, Vol. 7, pp. e769. DOI: 10.7717/peerj-cs.769.
13. **Chen, C., da-Silva, B., Chen, R., Li, S., Li, J., Liu, C. (2022).** Evaluation of fast sample entropy algorithms on FPGAs: From performance to energy efficiency. Entropy, Vol. 24, No. 9, pp. 1177. DOI: 10.3390/e24091177.
14. **Chen, H., Madaminov, S., Ferdman, M., Milder, P. (2020).** FPGA-accelerated samplesort for large data sets. Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, pp. 222–232. DOI: 10.1145/3373087.3375304.
15. **Chen, W., Li, W., Yu, F. (2020).** A hybrid pipelined architecture for high performance top-k sorting on FPGA. IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 67, No. 8, pp. 1449–1453. DOI: 10.1109/TCSII.2019.2938892.
16. **Coussy, P., Gajski, D. D., Meredith, M., Takach, A. (2009).** An introduction to high-level synthesis. IEEE Design and Test of Computers, Vol. 26, No. 4, pp. 8–17. DOI: 10.1109/MDT.2009.69.
17. **Diallo, A., Zopf, M., Fürnkranz, J. (2020).** Permutation learning via lehmer codes. 24th European Conference on Artificial Intelligence, IOS Press, pp. 1095–1102.
18. **Draz, H. H., Elashker, N. E., Mahmoud, M. M. A. (2023).** Optimized algorithms and hardware implementation of median filter for image processing. Circuits, Systems, and Signal Processing, Vol. 42, No. 9, pp. 5545–5558. DOI: 10.1007/s00034-023-02370-x.
19. **Esau-Taiwo, O., Christianah, A. O., Oluwatobi, A. N., Aderonke, K. A., Kehinde,**

A. J. (2020). Comparative study of two divide and conquer sorting algorithms: Quicksort and mergesort. Procedia Computer Science, Vol. 171, pp. 2532–2540. DOI: 10.1016/j.procs.2020.04.274.

20. **Fang, J., Mulder, Y. T. B., Hidders, J., Lee, J., Hofstee, H. P. (2019).** In-memory database acceleration on FPGAs: A survey. The VLDB Journal, Vol. 29, No. 1, pp. 33–59. DOI: 10.1007/s00778-019-00581-w.

21. **Gowanlock, M., Karsin, B. (2019).** A hybrid CPU/GPU approach for optimizing sorting throughput. Parallel Computing, Vol. 85, pp. 45–55. DOI: 10.1016/j.parco.2019.01.004.

22. **Hanafi, M. R., Faadhilah, M. A., Dwi-Putra, M. T., Pradeka, D. (2022).** Comparison analysis of bubble sort algorithm with tim sort algorithm sorting against the amount of data. Journal of Computer Engineering, Electronics and Information Technology, Vol. 1, No. 1, pp. 29–38. DOI: 10.17509/coelite.v1i1.43794.

23. **Ihaka, R., Gentleman, R. (1996).** R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics, Vol. 5, No. 3, pp. 299–314. DOI: 10.1080/10618600.1996.10474713.

24. **Jayaraman, S., Zhang, B., Prasanna, V. (2022).** Hypersort: High-performance parallel sorting on HBM-enabled FPGA. Proceedings of the International Conference on Field-Programmable Technology, pp. 1–11. DOI: 10.1109/ICFPT56656.2022.9974209.

25. **Kirkeby, M. H., Krabben, T., Larsen, M., Mikkelsen, M. B., Petersen, T., Rosendahl, M., Schoeberl, M., Sundman, M. (2022).** Energy consumption and performance of heapsort in hardware and software. arXiv, pp. 1–3.

26. **Kobayashi, R., Miura, K., Fujita, N., Boku, T., Amagasa, T. (2021).** A sorting library for FPGA implementation in OpenCL programming. Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, pp. 1–6. DOI: 10.1145/3468044.3468054.

27. **Kobayashi, R., Miura, K., Fujita, N., Boku, T., Amagasa, T. (2022).** An open-source FPGA library for data sorting. Journal of Information Processing, Vol. 30, pp. 766–777. DOI: 10.2197/ipsjjip.30.766.

28. **Kristo, A., Vaidya, K., Cetintemel, U., Misra, S., Kraska, T. (2020).** The case for a learned sorting algorithm. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1001–1016. DOI: 10.1145/3318464.3389752.

29. **Kuzmin, D., Tkachenko, M., Nikolaienko, A. (2023).** Visualization and analysis of sorting algorithms. Prospective Directions of Scientific Research in Engineering and Agriculture, pp. 74–81. DOI: 10.46299/isg.2023.mono.tech.1.3.2.

30. **Lade, S., Patil, K., Chhadikar, N., Chaudhari, A. (2019).** Effective sorting using parallel computing. International Journal of Advanced Research in Computer Engineering and Technology.

31. **Lobo, J., Kuwelkar, S. (2020).** Performance analysis of merge sort algorithms. Proceedings of the International Conference on Electronics and Sustainable Communication Systems, pp. 110–115. DOI: 10.1109/ICESC48915.2020.9155623.

32. **Moghaddamfar, M., Färber, C., Lehner, W., May, N. (2020).** Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA. Proceedings of the 16th International Workshop on Data Management on New Hardware, pp. 1–7. DOI: 10.1145/3399666.3399897.

33. **Montesdeoca, G., Asanza, V., Chica, K., Peluffo-Ordóñez, D. H. (2022).** Analysis of sorting algorithms using a WSN and environmental pollution data based on FPGA. Proceedings of the International Conference on Applied Electronics, pp. 1–4. DOI: 10.1109/AE54730.2022.9920090.

34. **Muthavarapu, A. K., Biswas, J., Barai, M. (2022).** An efficient sorting algorithm for capacitor voltage balance of modular multilevel converter with space vector pulsewidth modulation. IEEE Transactions on Power Electronics, Vol. 37, No. 8, pp. 9254–9265. DOI: 10.1109/TPEL.2022.3160665.

35. **Nikolajevic, K. (2016).** Dynamic autonomous decision-support function for piloting a helicopter in emergency situations. Ph.D. thesis, Laboratoire d'Automatique de Mécanique et d'Informatique industrielles et Humaines, Université de Valenciennes et du Hainaut-Cambresis, France.

36. **Ollivier-Legeay, H., Cadi, A. A. E., Belanger, N., Duvivier, D. (2020).** A 4D augmented flight management system based on flight planning and trajectory generation merging. Lecture Notes in Computer Science, pp. 184–195. DOI: 10.1007/978-3-030-63486-5_21.

37. **Rahul, G., Sandeep, P., Latha, Y. L. M. (2020).** Quicksort algorithm—an empirical study. Advances in Intelligent Systems and Computing, pp. 387–401. DOI: 10.1007/978-981-15-1480-7_33.

38. **Shinyamada, K., Yamawaki, A. (2021).** Effect of sorting algorithms on high-level synthesized image processing hardware. Proceedings of the 8th International Conference on Intelligent Systems and Image Processing, pp. 16–20.

39. **Soomro, I., Ali, H., Lashari, H. N., Maitlo, A. (2021).** Performance analysis of heap sort and insertion sort algorithm. International Journal of Emerging Trends in Engineering Research, Vol. 9. DOI: 10.30534/ijeter/2021/08952021.

40. **Souissi, O., Atitallah, R. B., Duvivier, D., Artiba, A. (2013).** Optimization of matching and scheduling on heterogeneous CPU/FPGA architectures. IFAC Proceedings Volumes, Vol. 46, No. 9, pp. 1678–1683. DOI: 10.3182/20130619-3-ru-3018.00196.

41. **Streit, M., Gehlenborg, N. (2014).** Bar charts and box plots. Nature Methods, Vol. 11, No. 2, pp. 117–117. DOI: 10.1038/nmeth.2807.

42. **Sunny, S. P., Narayanan-M, P. (2022).** Parallel sorting based OS-CFAR implementation in FPGA. Oceans, pp. 1–8. DOI: 10.1109/OCEANSChennai45887.2022.9775472.

43. **Zhang, T., Rahimi-Azghadi, M., Lammie, C., Amirsoleimani, A., Genov, R. (2023).** Spike sorting algorithms and their efficient hardware implementation: A comprehensive survey. Journal of Neural Engineering, Vol. 20, No. 2. DOI: 10.1088/1741-2552/acc7cc.

44. **Zhang, Z., Li, J. (2023).** A review of artificial intelligence in embedded systems. Micromachines, Vol. 14, No. 5, pp. 897. DOI: 10.3390/mi14050897.

Article received on 25/02/2024; accepted on 17/05/2024.
**Corresponding author is Yomna Ben-Jmaa.*