A COMPARISON-FREE SORTING ALGORITHM ON CPUs

Saleh Abdel-hafeez[1], Ann Gordon-Ross[2] and Samer Abubaker[3]
[1]*Jordan University of Science and Technology and Member IEEE, IRBID 22110, Jordan*
[2]*Department of Electrical and Computer Engineering, University of Florida (UF), affiliated with the NSF Center for High-Performance Reconfigurable Computing (CHREC) at UF and Member IEEE, Gainesville, FL 32611, USA*
[3]*Jordan University of Science and Technology, IRBID 22110, Jordan*

ABSTRACT

The paper presents a new sorting algorithm that takes input data integer elements and sorts them without any comparison operations between the data—a comparison-free sorting. The algorithm uses a one-hot representation for each input element that is stored in a two-dimensional matrix called a one-hot matrix. Concurrently, each input element is also stored in a one-dimensional matrix in the input element's integer representation. Subsequently, the transposed one-hot matrix is mapped to a binary matrix producing a sorted matrix with all elements in their sorted order. The algorithm exploits parallelism that is suitable for single instruction multiple thread (SIMT) computing that can harness the resources of these computing machines, such as CPUs with multiple cores and GPUs with large thread blocks. We analyze our algorithm's sorting time on varying CPU architectures, including single- and multi-threaded implementations on a single CPU. Our results show a fast sorting time for the single-threaded implementation that surpasses most common sorting algorithms with an average speedup of 6X for a number of input elements ranging from 2^3 to 2^{30}. Additional analysis using 8 threads with varying single-instruction-multiple-data (SIMD) widths shows an average speedup of 3.9X as compared to current parallel sorting algorithms for larger input sizes on the order of 2^{30} and higher.

KEYWORDS

Comparison-Free, Large Data, Parallel Computing, SIMD, Sorting Algorithms

1. INTRODUCTION

Sorting algorithms have been widely researched for decades [1]-[6] due to a ubiquitous need in many application domains [7]-[11]. Many algorithms have been specialized for particular sorting requirements/situations, such as large computations for processing data [12], high-speed sorting [13], special patterns of data types [14], sorting using a single CPU [15], exploiting the parallelism of multiple CPUs, parallel processing on high grid-computing in order to leverage the CPUs' powerful computing resources for big data processing [16], and recently using GPU platform computing resources for large high speed data processing [17]. Other works focus on architecting customized hardware designs for sorting algorithms in order to leverage the utilization of hardware resources and provide high-speed hardware processing [18]-[23]. However, due to the inherent complexity of sorting algorithms, efficient hardware implementation is challenging. To realize fast hardware sorting for big data, a significant amount of hardware resources are required. Even though Moore's law affords significant increases in chip transistor capacity, sorting is not the only computational requirement for applications, and thus the resources cannot be completely expended for sorting alone. Recent trends in improving sorting performance tailor the algorithms to leverage multi-core CPU computing resources, mainly due to the high degree of parallelism provided. Hence, data parallel codes are particularly suitable since the hardware can be classified as SIMT (single instruction, multiple thread). Much research has focused on harnessing the computational power of these resources for efficient sorting [24]-[27], however, there are still outstanding challenges for improving sorting algorithms to utilize parallel-processing units more efficiently. Most of the sorting algorithms depend on recursive comparison within particular input element partitions, and further requires collectively merging partitions in global memory. Even considering the plethora of prior work, there is no clear dominate sorting algorithm due to many factors [28]-[30], including the algorithm's percentage utilization of the available computing resources against memory resources, the specific data type being sorted, amount of data being sorted, etc.

As a result, since not all computing domains and sorting algorithms can leverage the high throughput of multi-core CPUs, there is still a great need for novel and transformative sorting methods. In this work, we propose a new sorting algorithm that processes *N* input data elements in linearly-separable vectors of one-hot weight representations. Each vector is multiplied individually with the input data elements to produce the sorted output using simple logic. We evaluated our proposed algorithm using a multi-core CPU with both single- and multi-threaded implementations by parallelizing our algorithm into an independent scalar product of linearly separable vectors, and directing each vector to a kernel-processing element. Leading to our algorithm's simplicity, the one-hot weight multiplication with the binary data is actually a single-gated binary data operation since, only 1 bit is multiplied with the input data, which reduces the multiplication operation to a simple switching data operation. Our proposed algorithm alleviates the use of a merging process (i.e., avoids using global memory for merging sorted partitions) since every column within the scalar multiplication releases the elements in the proper sorted order. Based on this design, if the algorithm performs these processes on the columns using separate processing elements, the algorithm's speed complexity is on the order of O(*N*), which makes our sorting method suitable for a wide range of sorting applications, and is competitive with state of the art sorting methods [31]-[33].

2. ALGORITHM PRINCIPLE

Our comparison-free sorting algorithm reduces the computational complexity by eliminating the comparison unit, and thus, avoiding the repetitive comparisons between elements, and data movements between the memory and the comparison units. The main computational paradigm of our algorithm is an array matrix operation, which is suitable and effective in utilizing parallel resources. The input to our sorting algorithm is a *K*-bit binary bus, which enables sorting *N*=2*K* distinct input data elements. Each element is represented with a one-hot weight representation, which is a unique count weight associated with each of the *N* elements. For example, "5" has a binary representation of "101", which has a one-hot weight representation of "100000". Therefore, for the complete set of *N*=2*K* distinct data elements, the complete representation contains all distinct binary elements of size one-hot weight *H*=*N*. For example, a *K*=3-bit input bus can sort/represent *N*=8 distinct elements, where each element's one-hot weight representation is of size *H*=8-bit (i.e., *H*=*N*). Continuing with our example, "5", with a binary representation of "101" has a one-hot weight representation of "00100000".

Our sorting algorithm requires two phases: the initialization phase to store the input data elements in an array (*BS*) of size *N*x1, where each element is of size *K*-bits. Concurrently, the input data elements are converted to the elements' one-hot weight representations and stored into a transpose memory (*TM*) of size *N*x*H*, where each stored element is of size *H*-bit and *H*=*N* giving a transpose memory of size *N*-bit x *N*-bit. The second phase, the evaluate phase, effectively sorts the elements by outputting the transposed elements using a matrix multiplication operation between *TM* and *BS*, rather than using comparison operations as in prior work. The multiplication operation is simplified to a switch operation since the size of each entry in the transpose memory is only 1-bit, which can be either "0" or "1". Subsequently, the data elements are read from the transpose memory, where each transposed row activates the element in *BS* into the sorted array (*SS*), which contains the final sorted data elements. Figure 1 depicts our comparison-free sorting using matrix multiplication based on linear algebra vector computations, including a simple illustrative example. This example shows our sorting algorithm's functionality using four input data elements of size 2-bit, with an initial (random) ordering of {2, 0, 3, 1}, which generates the outputted elements in *SS* = {3, 2, 1, 0}.

Duplicated data elements are represented using the same vector space, such that the corresponding transpose memory (*TM*) has multiple '1' values within a column of *TM*. The number of ones within the column of *TM* equals the number of times that data element is repeated in the input. These multiple 1s enable the same element in *BS* with no contention/confliction since these elements occupy a different index in *BS*. Consequently, when the column is read, the multiple ones within the associated column are mapped to the same elements in *BS*, which releases the same element every time this column is read in non-increasing order (i.e., the number of times the column is read is equal to the accumulated number of 1s within the entries in the read column). Additionally, for any column with only '0' value entries (i.e., the row's associated element is not in the input data), there is no associated read operation and the address pointer into the transpose memory is incremented to the next column in the sequence. In the best case, once each read operation processes a single data element, the read operation requires N iterations to generate the sorted output data for N input data elements. However, the worst case read operation occurs when all columns of the transpose memory have all '0' value entries except for the last column, which would have all '1' value entries. This case requires N-1 iterations, where no read operations occur due to all '0' values, plus N iterations for reading the last column, which has all value entries set to '1'. Thus the worst case read operation requires 2N-1 iterations.

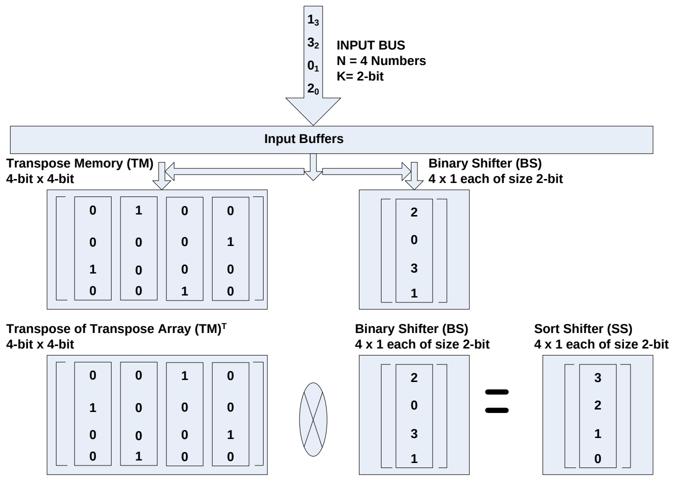

Figure 1. Comparison-free sorting example using 4 2-bit input data elements

Considering the sum of the two phases', the best case sorting time delay (lower bound) is 2N iterations when all N input data elements are unique. The worst case sorting time delay (upper bound) is 3N-1 iterations when all N input data elements are equal. These bounds are independent of the data type/representation being sorted or the input data's relative sequence/order. Thus, our sorting algorithm's complexity is on the order O(N) independent of the data type.

3. CPU MULTICORE COMPUTING

The proposed sorting algorithm is based on the mathematical algorithm depicted in Figure 1. Hence, the C-code program has two vector matrices: the array matrix (*BS*) of size Nx1 and the sorted matrix (*SS*) of size Nx1. *BS* stores the input data elements and *SS* stores the resulting sorted data elements. Additionally, the transpose one-hot matrix (*TM*) of size of NxN is converted to a one-dimensional matrix of size *Nx1* where each binary value is used as the index of a matrix and the count of the binary value is the matrix's stored value that is associated with that matrix index. Hence, the two-dimensional matrix is composed of only a one-dimensional matrix, and thus reduces the storage memory from NxN to Nx1, making the storage memory efficient and the operations fast when retrieving and storing data.

3.1 Single-threaded Implementation

The C-code of the initialization phase is illustrated in the first for loop in Figure 2, where the indices of *TM* record the input data elements of size Nx1. The elements in *BS* are read sequentially in the order that they appear in the input sequence (we assume for convenience that *BS* starts indexing at 0). With respect to *BS*, the code exhibits good spatial locality, but poor temporal locality since each vector element is accessed exactly once. Additionally, the elements in *TM* are referenced twice during each loop iteration, and exhibit good temporal locality with respect to the index vector *TM*. Overall, our sorting algorithm's initialization phase exhibits good spatial and temporal locality with respect to each variable in the loop body, which results in good performance when retrieving and updating data memory.

```
1.  Input:  Integer Element BS[0 : n - 1 ]
2.  Output: Integer Sort SS[0 : n - 1 ]
3.  One-Hot Weight: char TM[0:n-1] initialize to zero
4.  for i = 0 to n  do
5.      TM[E[i]] ←TM[E[i] ] + 1
6.    endfor
7.  Z ←0
8.  for i = 0 to n  do
9.      if TM[i]>0  then
10.             SS[Z] ←i
11.             Z←Z+1
12.             TM[i] ←TM[i]-1
13.             i←i−1
14.       endif
15. endif
```

Figure 2. Comparison-free sorting using C-code for a single-threaded, single CPU

```
1.  Input:  integer Element BS[0 : n - 1 ]
2.  Output: integer Sorted SS[0 : n - 1 ]
3.  Element Weight: char TM[0:n-1] initialize to zero
4.  Counter:  integer C[0 : nthS – 1] initialize to zero
5.    for i = 0 to n  do
6.          TM[BS[i]]←TM[BS[i]]+1
7.          C[BS[i]/(n/nthS)]←C[BS[i]/(n/nthS)]+1
8.    endfor
9.  do to all threads
10.     Z←sum of C[0] to C[nth-1]
11.     for i = nth*n/nthS to  (nth+1)*n/nthS   do
12.             if TM[i]>0 then
13.                     SS[Z]←i
14.                     Z←Z+1
15.                     TM[i]←TM[i]-1
16.                   i←i-1
17.             endif
18.       endfor
19.   end thread
key :
Number of Thread: nth:1.2.3.4……
Number of Total Threads: nthS
```

Figure 3. Comparison-free sorting using C-code using multi-threading of the second loop

The evaluation phase is illustrated in the second for loop in Figure 2, where the input data elements are sorted and stored in to the sorted vector *SS*. The elements in *TM* and *SS* are read and written sequentially, respectively, resulting in good spatial locality, which affords high performance when this locality is exploited by small storage memory with minor computations. Both loops read the elements of the array in row-major order; a characteristic that is more suitable as compare to column–major order for Ansi-c and the Gcc compiler. Additionally, both loops visit each element of a vector sequentially with a reference stride of one (with respect to the element size), which also exhibits efficiency for locality principles [34].

3.2 Multi-threaded

Our multi-threaded implementation exploits parallel computing power by partitioning our proposed sorting algorithm (Figure 1) into several parallel logical flows, where each flow can be assigned to a thread. The concurrent partitions of the algorithm are afforded by the inherent matrix computations and the independent mapping between the one-hot transpose rows in *TM* and the input *BS* array. Since context switching and atomic operations require more CPU time for scheduling and swapping data memory, we derive two structures that tradeoff more local storage memory for less context switching and atomic operation overheads to improve the sorting speed.

In reference to the single thread C-code algorithm in Figure 2, we parallelized the second loop structure such that every thread executes a partition (range) in *TM* with respect to the input *BS*. All threads execute the same thread routine, however, *SS* is shared and referenced by threads that are involved in the computations. One way to avoid the possibility of synchronization errors is to assign the index variable for *SS* as atomic, such that the index variable is incremented by only one thread and blocks all other threads' computations. However, using this atomic operation is not time/resource efficient since all threads must block except one thread. For more efficient operation we use a weighted counter variable in the initialization loop as illustrated in Figure 3, in order to assign a distinct range of input elements for each thread. Thus, every thread works independently on a partition of *SS*, as shown in the second loop of Figure 3, which avoids conflicts with repeated elements.

In the second version of our multi-threaded implementation, we parallelize the first and second loop structures, as shown in Figure 5. In this structure, each thread has its own weighted counter variable in multiples of the thread variables, and each thread has it own *Q* array vector, which stores the sorting results for that thread's partition. Additionally, we insert another loop between the two loops in Figure 1 to merge all threads' *Q*s to only one *Q*, and merge the multiple weight counters in each thread to only one weight counter per thread. The second loop in Figure 5 is the same as in Figure 4. Our results show that this approach produces the highest speed considering the cost of the large storage memory structure for each thread.

```
1.  Input:  Integer  Element BS[0 : n - 1 ]
2.  Output: Integer Sort SS[0 : n - 1 ]
3.  Element Weight: char TM[0:n-1]initialize to zero
4.  Counter: integer  C[0 : nthS - 1][0 : nthS - 1]initialize to zero
5.  do to all threads
6.          for i = nth*n/nthS  to (nth+1)*n/ nthS  do
7.                  TM[nth][BS[i]]← TM[nth][BS[i]]+1
8.                  C[nth][BS[i]*nthS/n]← C[nth]
                            [BS[i]*nthS/n]+1
9.          endfor
10.     barrier wait all thread
11.         for i = nth*n/nthS  to (nth+1)*n/ nthS  do
12.             TM[0][i]← sum of TM[0][i] to TM[nth][i]
13.         endfor
14.     barrier wait all thread
15.         Z← sum of C[0][0] to C[nthS][nth-1]
16.         for i = nth*n/nthS  to (nth+1)*n/ nthS  do
17.             if  TM[0][i] > 0  then
18.                     SS[Z]←i
19.                     Z←Z+1
20.                     TM[0][i]←TM[0][i]-1
21.                     i← i-1
22.             endif
23.         endfor
24. end thread
key :
Number of Thread: nth:1.2.3.4......
Number of Total Threads: nthS
```

Figure 4. Comparison-free sorting using C-code using multi-threading on the both loops

4. SIMULATION RESULTS AND COMPARISON

In this section, we evaluate the performance of our comparison-free sorting algorithm using varying input sizes, where each input size is a power of 2. We sort integer input data for different input-set-ordering scenarios, including a uniform random distribution, reverse ordering, nearly-sorted, and a few unique elements that are repeated in the input set for a thorough evaluation. We implemented our sorting algorithm in the Linux operating system using the GCC compiler, and executed our simulations on an Intel CoreTM CPU I7-3770 with a 3.4 GHz processor, 8 GB of RAM, and 8 MB of cache.

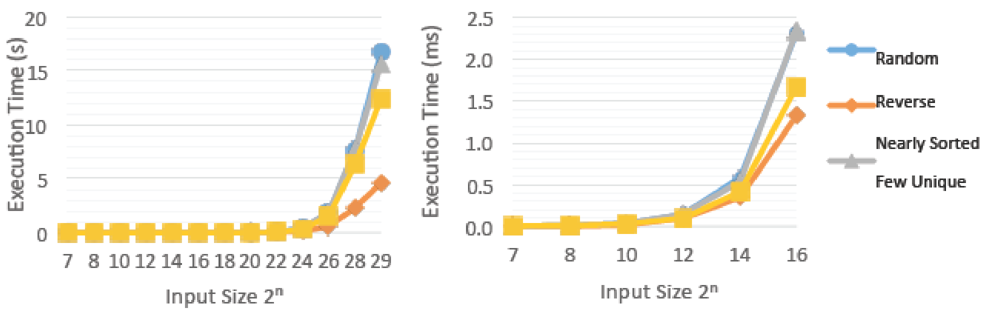

Figure 5. Comparison-free sorting execution time for the single-threaded C code for varying input sizes and input set orderings (left) and a subset of the input sizes ranging from 2^7 to 2^{16} to zoom in and show details of the smaller input sizes (right)

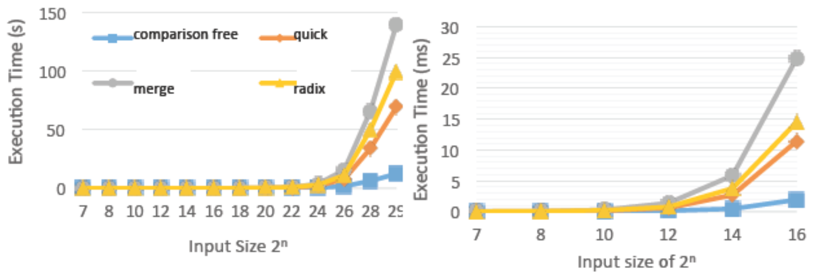

Figure 6. Comparison-free sorting execution time for the single-threaded C code for varying input sizes as compared to current sorting algorithms (left) and a subset of the input sizes ranging from 2^7 to 2^{16} to zoom in and show details of the smaller input sizes (right)

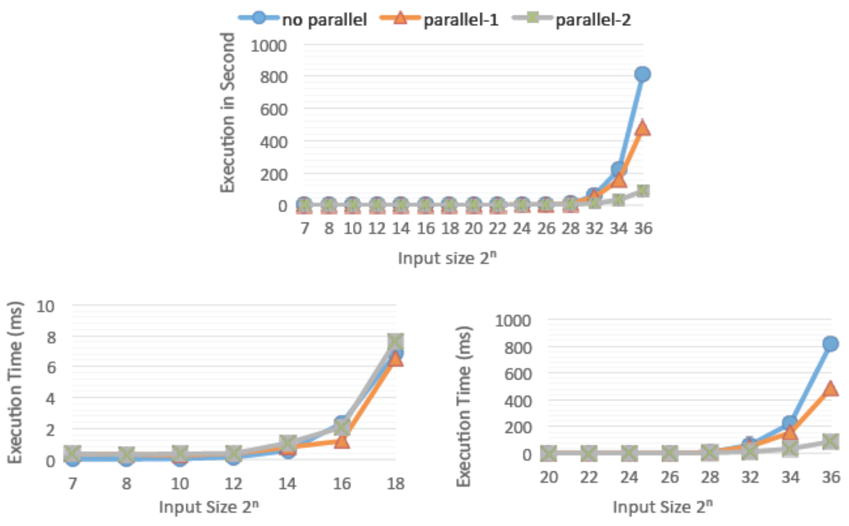

Figure 7. Comparison-free sorting execution time for single-threaded (no parallel) and multi-threaded (parallel-1, parallel-2) implementations for varying input set sizes (top) and a subset of the input set sizes to zoom in on the details (bottom)

We report the actual sorting execution time in seconds, and take into account all memory copies and contention with context switching times. Figure 5 reports the sorting times using the C-code derived in Figure 2, which is our proposed single-threaded sorting algorithm. The left graph depicts the entire range of input sizes, and the right graph zooms into the smaller sizes to show details, and presents the results in milliseconds. The figure illustrates fast execution times for very large input sets, with little difference in input order. The execution time does not increase appreciably until the input data set size is 2^{28} = 268,435,456 elements. Figure 6 (left and right layout similar to Figure 5) compares our sorting algorithm to other popular sorting algorithms, and shows considerable execution time reductions for sorting large input set sizes, with minimum execution time reductions of approximately 6X for input data ranges between 16 and 64,000,000,000 elements, making our algorithm one of the fastest sorting algorithms as compared to current work, to the best of our knowledge.

We evaluated our algorithm’s parallel execution time for an 8-threaded simulation in Figure 7 as compared to the single-threaded (no parallel) and multi-threaded (parallel-1 and parallel-2) implementations using the C-code derived in Figure 3 and Figure 4, respectively. These results show the effectiveness of parallelism for large input set sizes, resulting in high speed sorting. Figure 7 shows all input set sizes in the top graph, and the bottom graphs zoom into two input set size ranges to show the details more clearly. Results show that the multi-threaded implementation is about 3.9X faster on average as compared to the single-threaded implementation for input set sizes greater than 256,000,000 elements. Additionally, these results show higher performance for the C-code derived in Figure 4 as compared to the C-code derived in Figure 3, however, we point out that the C-code derived in Figure 4 provides additional speedup at the cost of large memory structures per thread.

Table 1. Sorting time in second for different algorithms that are parallelized on a Core 2 Quad CPU

Algorithms vs. Input Set Size	**2^{20}**	**2^{22}**	**2^{24}**	**2^{26}**	**2^{28}**
Butterfly [35]	0.6	3	8.6	18.1	33.7
Radix Sort [36]	0.44	1.7	6.7	27.1	83.2
AA-SORT [15]	0.47	0.36	0.9	4.8	17
Proposed Parallel-1	0.014	0.049	0.425	1.88	8.11
Proposed Parallel-2	0.017	0.058	0.234	1.08	4.41

Table 1 compares the execution time of our parallel algorithm to other well-known parallel sorting algorithms for large input set sizes. These results show the performance advantage of our algorithm as compared to other existing algorithms, which is mainly due to the parallel nature of our algorithm's vector operations with minimal arithmetic operations (i.e., only one IF-statement for the mapping function). In addition, our algorithm inherits small storage memory requirements (parallel-1) that are comprised of vector arrays rather than two-dimensional arrays.

5. CONCLUSIONS AND FUTURE WORKS

In this paper, we proposed a novel comparison-free sorting algorithm, and associated software implementations for single- and multi-threaded implementations. To the best of our knowledge, our design is the first to leverage the data input element's one-hot weight representation in conjunction with the element's binary representation to sort input data element sets without any comparison operations, and using only a simple matrix mapping operation. The data computation flows in a forward-flowing direction (i.e., each element is evaluated only once) through the data path without any arithmetic logic unit (ALU) processing components (i.e., no comparison operations. Thus, one of the major advantages of our sorting algorithm is that our design alleviates all power associated with the comparison operation, which can be a significant percentage of the power consumption. Due to the computations' simplicity, the software implementations with no-parallel and parallel data flows show performance improvements as compared to well-known sorting algorithms. Simulations for the single-threaded implementation show that our algorithm reduces the execution time as compared to quicksort3 by an average of 6X for input set sizes up to 2^{30} = 1,073,741,824 elements, and simulations for the multi-threaded (8-threads) implementation reduces the execution time as compared to AA-Sort by an average of 3.9X for input set sizes up to 2^{30} = 1,073,741,824 elements.

Future work includes leveraging our sorting algorithm to commercially existing parallel processing computing power, such as GPUs and parallel processing machines, in order to further extend performance advantages on big data and further reduce adverse memory effects, and thus, further enhance the processing time for big data.

REFERENCES

[1] Donald E. Knuth, 2011. *The Art of Computer Programming*, Addison-Wesley Professional.

[2] Yanjun Bang and S.Q. Zheng, 1994. "A Simple and Efficient VLSI Sorting Architecture", *Proceedings of the 37th Midwest Symposium on Circuits and Systems*, Vol. 1, pp.70-73.

[3] Tom Leighton and Yuan Ma, 1997. "Breaking the O(n log2n) Barrier for Sorting with Faults", *Journal of Computer and System Sciences*, Vol. 54, pp. 265-304.

[4] Yijie Han, 2004. "Deterministic sorting in O(nloglogn) time and linear space, Journal of Algorithms", *Science Direct publisher*, Vol. 50, pp. 96-105.

[5] C. Canaan, M. S. Garai, and M. Daya,2011. "Popular Sorting Algorithms", *World Applied Programming*, Vol. 1, No. 1, pp. 62-71.

[6] L. M. Busse, M. H. Chehreghani, J. M. Buhmann, 2012. "The Information Content in Sorting Algorithms", *IEEE International Symposium on Information Theory Proceedings (ISIT)*, pp. 2746-2750.

[7] Ran Zhang, Xue Wei and Takahiro Watanabe, 2013. "A Sorting-Based IO Connection Assignment for Flip-Chip Designs", *2013 IEEE 10th International Conference on ASIC (ASICON)*, pp. 1-4.

[8] Dong Fuguo, 2010. "Several Incomplete Sort Algorithms for Getting the Median Value", *International Journal of Digital Content Technology and its Applications*, Volume 4, Number 8, pp. 193-198.

[9] Wu Jianping, Ye Yutang, Liu Lin, Huang Bingquan, Guo Tao, 2011. "High-speed FPGA-based SOPC Application For Currency Sorting System", *The Tenth International Conference on Electronic Measurement & Instruments (ICEMI'2011)*, pp. 85-89.

[10] Robert Meolic, 2013."Demonstration of Sorting Algorithms on Mobile Platforms", *CSEDU SciTePress*, pp.136-141.

[11] Fang-Cheng Leu, Yin-Te Tsai, Chuan Yi Tang, 2000. "An efficient external sorting algorithm", *2000 Elsevier Science*, Information Processing Letters 75, pp. 159–163.

[12] Jon L. Bentley and Robert Sedgewick, 1997. "Fast Algorithms for Sorting and Searching Strings", *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pp. 360-369.

[13] Li Xiao, Xiaodong Zhang, Stefan A. Kubricht,, 2000." Improving Memory Performance of Sorting Algorithms", *ACM Journal of Experimental Algorithmic*, Vol. 5, No. 3, pp. 1-20.

[14] Pankaj Sareen, ,2013. "Comparison of Sorting Algorithms (On the Basis of Average Case)", *IJARCSSE,Vol*. 3, issue 3, pp.522-532.

[15] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu and Toshio Nakatani, 2007. "AA-SORT: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors", *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 189-198.

[16] Vamsi Kundeti and Sanguthevar Rajasekaran, 2011."Efficient out-of-core sorting algorithms for the Parallel Disks Model", *Journal Parallel Distributed Computer*, Vol. 71, pp. 1427-1433.

[17] Gabriele capannini, Fabrizio Silvestri and Ranieri Baraglia, 2012. "Sorting on GPUs for large scale datasets: A through Comparison*", International Processing and Management* Vol. 48, pp. 903-917.

[18] Daniel Cederman and Philippas Tsigas, 2009."GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors", *ACM Journal of Experimental Algorithmics (JEA)*, Vol. 14, No. 4, Pages 1-22.

[19] Bilal Jan, Bartolomeo Montrucchio, Carlo Ragusa, Fiaz Gul Ghan and Omar Khan, 2012. "Fast Parallel Sorting Algorithms on GPUs", *International Journal of Distributed and Parallel Systems (IJDPS)* Vol. 3, No.6, pp.107-118.

[20] Nadathur Satish and Mark Harris and Michael Garland, 2009. "Designing Efficient Sorting Algorithms for Manycore GPUs*", 23rd IEEE International symposium on Parallel and Distributed Processing*, pp. 1-10.

[21] Christian Bunse and Hagen HÖpfner and Suman Roychoudhury and Essam Mansour, 2009. "Choosing the BEST Sorting Algorithm from Optimal Energy Consumption*", ICSOFT 2*, INSTICC Press, pp. 199-206.

[22] Aditya Dev Mishra and Deepak Garg, 2008. "Selection of Best Sorting Algorithm", *International Journal of Intelligent Information Processing*, Vol. 2, pp. 363-368.

[23] Tzu-Chin Lin, Chung-Chin Kuo, Yong-Hsiang Hsieh, Biing-Feng Wang, 2009. "Efficient algorithms for the inverse sorting problem with bound constraints under the L∞-norm and the Hamming distance", *Journal of Computer and system Sciences*, Vol. 75, pp. 451-464.

[24] Fritz Henglein, 2009. "What is a Sorting Function?", *The Journal of Logic and Algebraic Programming*, Vol. 78, Issue 7, pp. 552–572.

[25] Yanjun Zhang and S.Q. Zheng, 1994. "A Simple and efficient VLSI Sorting Architecture", *Proceedings of the 37th Midwest Symposium on Circuits and Systems*, Vol. 1, pp. 70-73.

[26] Enzo Mumolo, Gabriele Capello, and Massimiliano Nolich, 2004. "VHDL Design of a Scalable VLSI Sorting Device Based on Pipelined Computation", *Journal of Computing and Information Technology*, Vol. 12, pp. 1-14.

[27] Ezequiel Herruzo, Guillermo Ruiz, J. Ignacio Benavides, and Oscar Plata, 2007. "A new Parallel Sorting Algorithm based on Odd-Even Mergesort", *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 07)*, pp. 18-22.

[28] Mikkel Thorup, 2002. "Randomized Sorting in O(n log log n) Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations", *Journal of Algorithms*, Vol. 42, issue 2, pp. 205–230.

[29] M. Afghahi,1991. "A 512 16-b Bit-Serial Sorter Chip", *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 10, pp. 1452-1457.

[30] Jin-Tai Yan, 1999. "An Improved Optimal Algorithm for Bubble-Sorting Based Non-Manhattan Channel Routing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 2, pp. 163-171.

[31] Louliia Skliarova , Dmitri Mihhailov, Valery Sklyarov, and Alexander Sudnitson, 2012. "Implementation of Sorting Algorithms in Reconfigurable Hardware", *Electrotechnical Conference (MELECON), 16th IEEE Mediterranean*, pp. 107-110.

[32] Nozar Tabrizi and Nader Bagherzadeh, 2005. "An ASIC Design of a Novel Pipelined and Parallel Sorting Accelerator for a Multiprocessor-on-a-Chip", *IEEE 6th International Conference On ASIC (ASICON)*, pp. 46-49.

[33] H. SCHRÖDER, 1988. "VLSI-Sorting Evaluated under the Linear Model", *JOURNAL OF COMPLEXITY*, Vol. 4, Issue 4, pp. 330-355.

[34] R. E. Bryant and D. R. O'Hallaron,2003.*"Computer Systems: A programmer's Perspective"*,Pearson Education, Inc.

[35] B. Jan, B. Montrucchio, C. Ragusa, F. Khan, and O. Khan, 2012. "Fast Parallel Sorting Algorithms on GPUs*", International Journal of Distributed and Parallel Systems (IJDPS)*, Vol. 3, No. 6, pp. 107-118.

[36] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey, 2010. "Fast sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort", *SIGMOD '10*, Indiana, 27(3), pp. 351-362.