Bit-index Sort: A Fast Non-comparison Integer Sorting Algorithm for Permutations

L.F. Curi-Quintal[1,2]
[1]Faculty of Mathematics
University of Yucatan (UADY)
Merida, Mexico
l.f.curiquintal@pgr.reading.ac.uk

J.O. Cadenas
[2]School of Systems Engineering
University of Reading
Reading, United Kingdom
o.cadenas@reading.ac.uk

G.M. Megson
School of Electronics and Computer Sciences
University of Westminster
London, United Kingdom
g.megson@westminster.ac.uk

***Abstract*— This paper describes a fast integer sorting algorithm, herein referred to as Bit-index sort, which does not use comparisons and is intended to sort partial permutations. Experimental results exhibit linear complexity order in execution time. Bit-index sort uses a bit-array to classify input sequences of distinct integers, and exploits built-in bit functions in C compilers, supported by machine hardware, to retrieve the ordered output sequence. Results show that Bit-index sort outperforms quicksort and counting sort algorithms when compared in their execution time. A parallel approach for Bit-index sort using two simultaneous threads is also included, which obtains further speedups of up to 1.6 compared to its sequential case.**

Keywords*— sorting algorithm, permutations, linear complexity, bit-array index, counting leading zeros, counting trailing zeros.

I. Introduction

Sorting is a well known and widely investigated computing problem. In general, a sorting algorithm generates, from an input sequence of n values, $(v_1, v_2, ..., v_n)$, an ordered output sequence of those values, $(v'_1, v'_2, ..., v'_n)$, either increasingly $(v'_1 \leq v'_2 \leq ... \leq v'_n)$ or decreasingly $(v'_1 \geq v'_2 \geq ... \geq v'_n)$ [1]. Comparison sorting algorithms, such as quicksort [2] and heapsort [3], determine the ordered sequence of the input data elements based on comparison operations between such elements. Non-comparison sorting algorithms, such as counting sort [4] and radix sort [5], produce the ordered output making some suppositions about the size and range of the input data values and distribute them in intermediate data structures to gather and place them in the ordered output sequence.

This paper introduces an integer sorting algorithm, herein referred to as Bit-index sort, and argues it is a fast non-comparison sorting algorithm for sequences of integer numbers without repeated elements, i.e. for partial permutations. In combinatorics, a permutation of length n is a bijection from the set $D=\{1,2,...,n\}$ to itself. A partial permutation for the set $D=\{1,2,...,n\}$ is a sequence with k distinct elements of D, with $k<n$ [6].

Bit-index sort performs two stages: classification and retrieval. It is thus similar to counting sort, with the difference that it uses only an additional bit array to map the input elements in order to obtain the sorted output sequence in linear complexity order. Indexing schemes has been previously reported to accelerate sorting algorithms of quadratic complexity, applied to natural numbers [7], by inserting each input data element in an index array, and retrieving without comparison or swapping.

The sorting algorithm presented in [8] performs also two stages: mapping and post-mapping. This algorithm uses a guessing function to map each input element in its corresponding position in the output sequence, and performs local quicksort routines in cases when the guessing function allocates more than one element in the same position. It claims to get a complexity order in execution time between $O(n)$ and $O(n \log_2 n)$, but relies in the effectiveness of the guessing function to get a uniform distribution of elements in the output sequence.

Section II of this paper describes the Bit-index sort algorithm using an example. A software implementation in C language is explained in Section III and includes a parallel approach for two simultaneous processors. Sequential and parallel versions of Bit-index sort were compared with quicksort and counting sort implementations using two different ranges of integers and several sizes of input data sequences. Section IV exhibits these experimental results. Bit-index sort implementations present a linear complexity order in execution time and outperform quicksort and counting sort as Section V describes. Also, a performance analysis for the parallel approach of Bit-index sort is included. Bit-index sort complexity order is not better than other known algorithms, but our implementation is faster due to the principle of operation of the algorithm.

II. Bit-index Sort

Bit-index sort algorithm is performed in two main stages: *(i)* classification and *(ii)* retrieval, using a bit array as index. Let D be the input sequence with n distinct integer numbers with values in the interval $[0, m-1]$, and $n<m$. Let the bit-index, BI, be a bit array with m elements $[0...m-1]$ initialized with zero value. In the classification stage, the value of each element in D represents the position of bit element in BI which will be set, i.e. $\forall i \in D, BI[D[i]] = 1$. Fig. 1 includes a description for the classification algorithm.

After classifying all elements in D, bits set in BI determine the ordered output sequence for D. Using the equivalent unsigned integer to bit-array BI, the retrieval algorithm in Fig. 1 obtains the position of each bit set in BI using n loop steps starting on the least significant bit (LSB).

```
Classification algorithm
Input:
  Data <-- Array of n integers in [0, m-1]
Output:
  BI <-- Array of m bits

1. for each element i in Data
2.   BI[D[i]]=1
3. endfor

Retrieval algorithm
Input:
  BI <-- Bit-array as an unsigned integer
Output:
  Data <-- Array of ordered data

1. i=0
2. while (BI > 0)
3.    t = BI & (BI-1)
4.    Data[i] = log2(BI-t)
5.    i++
6.    BI=t
7. endwhile
```

Fig. 1. Pseudocode of classification and retrieval stages in Bit-index sort.

Fig. 2 displays an example of both stages of Bit-index sort for the data input sequence *D = (9, 6, 0, 4, 13, 11, 14, 1, 7, 12)* with 10 elements with values in the interval [*0, 15*]. Therefore, *BI* is a bit-array with 16 elements initialized with zero value. Section *(i)* in Fig. 2 shows the classification stage where each element in *D* represents the position of the bit in *BI* which will be set. After all the correspondent bits are set, the unsigned 16-bit integer value for *BI* is 31443. This value is used as input for the retrieval algorithm used in section *(ii)* of the figure. Table I exhibits the retrieval process for the ordered output sequence in *D*, using only 10 loop steps.

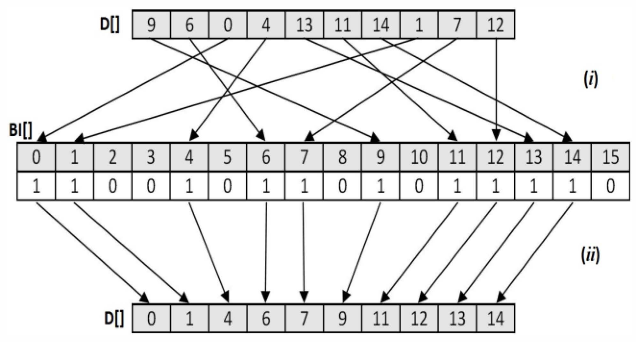

Fig. 2. Example of Bit-index sort.

Using the equivalent unsigned integer of *BI* apparently limits the range of data values to the size in bits of the largest unsigned integer available in the architecture and programming language used in the implementation. However, an array of unsigned long integers can be used as the bit-array for large ranges in data values.

For example, suppose *D* is an input data sequence with input values in [*0, 127*], and *BI*, the bit-index array, is an array of four 32-bits unsigned integers, which represent the 128 possible input values.

TABLE I. RETRIEVAL OF THE ORDERED OUTPUT SEQUENCE FROM BI IN FIG. 2 WITH 10 LOOP STEPS

i	***BI* as unsigned integer**	***t=BI & (BI-1)***	***D[i]=log$_2$(BI-t)***
1	31443	31442	0
2	31442	31440	1
3	31440	31424	4
4	31424	31360	6
5	31360	31232	7
6	31232	30720	9
7	30720	28672	11
8	28672	24576	12
9	24576	16384	13
10	16384	0	14

In the classification stage, each input data in *D* is processed by a module function and an integer division using the unsigned integer bit width (in this case is 32) as denominator in both operations. The results obtained from these operations represent the bit to be set and the correspondent integer element in *BI*, respectively, e.g. for an input value of *115*, the bit to be set will be the *19th* (*115 mod 32 = 19*) in the unsigned integer *BI[3]* (*115 / 32 = 3*). The correspondent bit is set using the logical operator OR with two operands: *(i)* the current integer value and *(ii)* the bit 1 left-shifted the number of positions given by the reminder obtained in the module operation. In the example, the operation *BI[3] = BI[3] .OR. (1<<19)* is computed in order to set the corresponding bit to the data value *115*.

After classifying all input data values, the retrieval stage can be applied on each element of the integer array *BI*, and the global position of each bit is obtained adding a global offset represented by the element position in *BI* multiplied by the element bit width (in the example is 32) plus the local bit offset in this element. In the example, the bit set in position *21* of the element *BI[2]* represents the global value of *85 (2*32+21 = 85)*.

III. IMPLEMENTATION IN C

Bit-index sort was implemented in C language using an array of unsigned long integers (64 bits size) as the bit-index array. The retrieval stage was implemented using two bit functions known as *count trailing zeros* (*ctz*), which counts the bits with zero value after the least significant bit set, and *count leading zeros* (*clz*) which counts the number of bits with zero value before the most significant bit set. Microprocessor manufacturers, like Intel[9], IBM[10], AMD[11] and ARM[12], have included hardware support in their chips for either one or both of these functions in order to obtain better performance. Also, these bit functions are available in the most popular C/C++ compilers, like GCC[13], Microsoft Visual Studio[14], Intel C/C++ Compiler[15] and NVIDIA CUDA[16], either as built-in functions or library functions.

Pseudocode in Fig. 3 implements Bit-index sort with retrieval stage using *ctz()* function to obtain the output sequence in ascending order starting from the least significant bit. Bit-index array is an array of unsigned 64-bits integers, initialized with zero value, that divides the global range of input values in segments of 64 bits each. Therefore, the classification stage consists of computing the segment (*ind*) and bit offset (*off*) for each input value in order to set the correspondent bit using the logical operator OR (lines 5-9). The retrieval stage obtains each bit set using *ctz()* and calculates the global position of the bit, which represent the output value, using the position of the unsigned integer in the array and the bit offset (lines 12-19).

```
Bin-Index ASC Algorithm
Input:
  Data <-- Array of integers
  N <-- Size of Data
  M <-- Maximum value of data integers
Output:
  Data with sorted elements

1. Bidx[0..M/64]={0} //Bit-index (unsigned int)
2. ind=0;     //index
3. off=0;     //offset
4. //Classification stage
5.  forall i in Data
6.    ind=i/64
7.    off=i%64
8.    Bidx[ind]=(Bidx[i]) .OR. (1<<off)
9.  endfor
10. //Retrieval stage
11. i=0
12. forall j in Bidx ascending
13.    while (Bidx[j]>0)
14.       off=ctz(Bidx[j])
15.       Data[i]=j*64+off
16.       Bidx[j]-=(1<<off)
17.       i++
18.    endwhile
19. endfor
```

Fig. 3. Pseudocode for ascending order output using Bit-index sort with *ctz()* function

Similarly, pseudocode in Fig. 4 implements Bit-index sort with retrieval stage using *clz()* function to obtain the output sequence in descending order starting from the most significant bit. Classification stage (lines 5-9) and retrieval stage (lines 12-19) are similar to pseudocode using *ctz()*. Either of bit functions, *ctz()* or *clz()*, can be used to obtain both ascending and descending order output sequence by allocating retrieved elements to the output array in ascending or descending order of its index.

Quicksort and counting sort algorithms were implemented in order to compare execution times with Bit-index sort versions. Quicksort was implemented using the library function *qsort()* in C language [17], which implements the quicksort algorithm with complexity order $O(n \log_2 n)$ [1]. Counting sort was implemented based on the algorithm presented in [1], with $O(n)$ complexity order. In principle, Bit-index sort algorithm applied to an input data sequence with n elements with values in $[0, m-1]$ with $n<m$, has a complexity $O(n+m)$, but as each input integer has at most $w = \log_2 m$ bits, then $O(n+m) = O(n + 2^w) = O(n)$. Therefore, Bit-index sort is considered a linear complexity order algorithm, with minor differences observed for sorting the average, best and worst cases.

Versions of Bit-index sort were implemented in gcc 4.4.4 compiler using built-in functions **__builtin_ctzl** (*unsigned long int*) for *ctz()*, and **__builtin_clzl** (*unsigned long int*) for *clz()* [13].

```
Bin-Index DSC Algorithm
Input:
  Data <-- Array of integers
  N <-- Size of V
  M <-- Maximum value of data
Output:
  Data with sorted elements

1. Bidx[0..M/64]={0}   //Bit-index (unsigned int)
2. ind=0;     //index
3. off=0;     //offset
4. //Classification stage
5.  forall i in Data
6.    ind=i/64
7.    off=i%64
8.    Bidx[ind]=(Bidx[i]) .OR. (1<<off)
9.  endfor
10. //Retrieval stage
11. i=0
12. forall j in Bidx descending
13.    while (Bidx[j]>0)
14.       off=63-clz(Bidx[j])
15.       Data[i]=j*64+off
16.       Bidx[j]-=(1<<off)
17.       i++
18.    endwhile
19. endfor
```

Fig. 4. Pseudocode for descending order output using Bit-index sort with *clz()* function

Furthermore, a parallel version of Bit-index sort was implemented given the availability of both bit functions (*ctz* and *clz*) in the utilised compiler. This parallel approach implements the retrieval stage using two simultaneous threads: one thread extracts output data starting with the first element of the Bit-index array using retrieval with *ctz()*, and the other thread extracts output data starting with the last element of the Bit-index array using retrieval with *clz()*. This approach was implemented using two parallel sections in OpenMP [18], one section for retrieval with *ctz()* and the other section for retrieval with *clz()*, computing one half of the output sequence each.

IV. Experimental Results

Bit-index sort implementations were compiled with gcc 4.4.4 (Red Hat 4.4.4-13) and executed on an Intel 12-core (dual 6-core Xeon X5690, 3.47 GHz) desktop system with Scientific Linux 6.0 (release 2.6.32-131).

Table II exhibits the results for execution times (in seconds) using the interval [0, 76799] (*M* in the algorithm) for random input values, different amounts (between 50 and 55000) for the number of elements in the input sequence (*N* in the algorithm), having up to 70% of the global bit-array set, and each execution was replicated one million times. Fig. 5 shows normalized speedups in execution times of the sorting algorithms with respect to quicksort results.

Table III exhibits the results for execution times (in seconds) using the interval [0, 307199] (*M* in the algorithm) for random input values, different amounts (between 500 and 182000), for the number of elements in the input sequence (*N* in the algorithm), having up to 60% of the global bit-array set, and each execution was replicated 1 million times. Fig. 6 shows normalized speedups in execution times of the sorting algorithms with respect to quicksort results.

TABLE II. EXECUTION TIMES, IN SECONDS, FOR SORTING ALGORITHMS WITH INPUT VALUES IN [0, 76799]

N	Q-sort	C-sort	BI-sort (clz)	BI-sort (ctz)	BI-sort parallel
50	2.6	201.8	3.5	3.4	2.6
100	5.9	202.5	4.4	4.4	3.2
250	16.9	203.5	7.3	7.1	5.0
400	29.0	204.9	10.0	9.7	6.8
700	54.4	207.1	15.0	14.8	10.2
1100	91.4	211.5	20.9	20.9	16.0
1600	137.4	213.2	27.3	27.3	20.5
2300	208.7	219.5	35.8	35.8	26.6
3000	275.0	226.5	44.2	44.3	32.6
4500	439.3	239.7	61.6	61.8	45.8
6700	679.5	260.5	87.1	86.4	64.3
9200	964.5	283.3	115.5	112.3	85.4
13100	1409.3	325.8	158.5	152.7	117.3
16400	1775.0	366.1	195.1	187.7	142.6
23500	2648.6	458.8	272.9	261.4	204.3
31700	3678.7	573.1	363.5	347.1	269.1
42400	5097.2	751.8	458.6	431.6	355.3
48900	5855.4	850.8	554.5	525.9	415.2
55000	6767.2	954.6	620.8	589.5	461.5

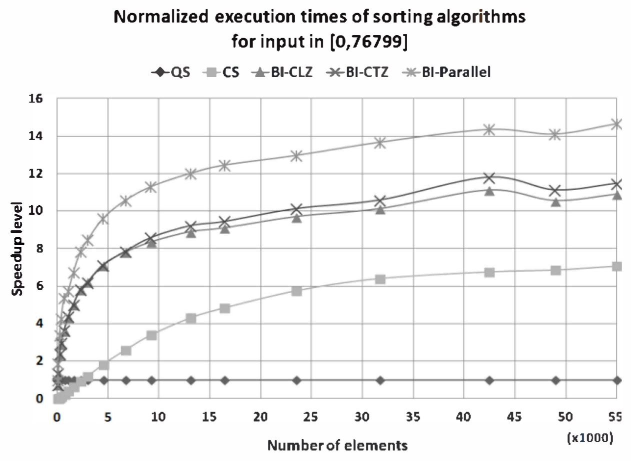

Fig. 5. Normalized execution times from values in Table II

TABLE III. EXECUTION TIMES, IN SECONDS, FOR SORTING ALGORITHMS WITH INPUT VALUES IN [0, 307199]

N	Q-sort	C-sort	BI-sort (clz)	BI-sort (ctz)	BI-sort parallel
500	37.3	889.1	20.0	19.5	11.7
1,000	81.5	898.3	29.7	28.8	17.9
2,000	176.5	917.4	47.4	46.3	30.0
4,000	381.2	957.1	79.6	78.5	51.8
7,000	712.6	1017.4	119.3	117.8	79.2
11,000	1164.5	1096.3	167.8	166.9	114.3
16,000	1748.3	1199.2	227.8	226.4	158.6
22,000	2485.0	1323.2	299.1	296.2	210.8
29,000	3364.8	1463.9	382.2	373.1	270.2
37,000	4413.0	1656.1	473.6	458.2	339.0
46,000	5533.5	1810.5	574.9	552.8	415.3
56,000	6896.7	2040.1	688.0	658.2	497.7
67,000	8293.2	2225.6	811.8	774.2	587.8
79,000	9998.2	2457.1	949.6	898.0	689.5
92,000	11751.4	2702.5	1096.6	1035.8	796.6
116,000	15124.0	3212.3	1374.1	1293.2	994.7
131,000	17056.1	3503.0	1544.8	1454.8	1119.1
147,000	20239.0	3801.9	1725.0	1617.2	1255.0
164,000	21828.2	4065.9	1922.1	1800.3	1396.8
182,000	24550.8	4456.4	2122.6	1990.0	1548.9

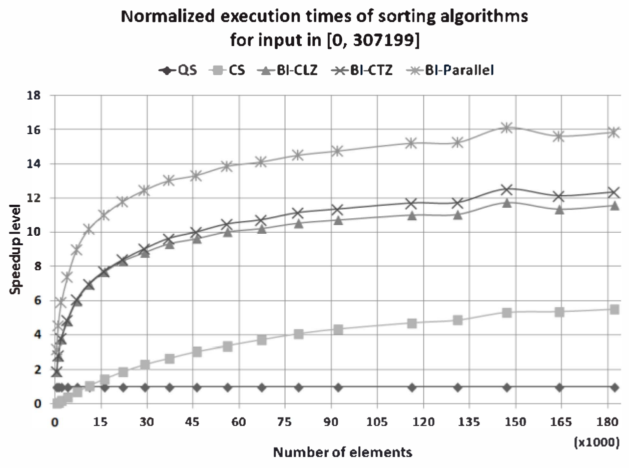

Fig. 6. Normalized execution times from values in Table III

In Fig. 7, the expected linear trend in execution time for Bit-index sort is revealed when corresponding values in the last three columns in Table III are plotted. These values represent execution times for Bit-index sort using *clz()*, *ctz()* and both functions in parallel, respectively. A similar graph is obtained when execution times in Table II for such implementations are plotted.

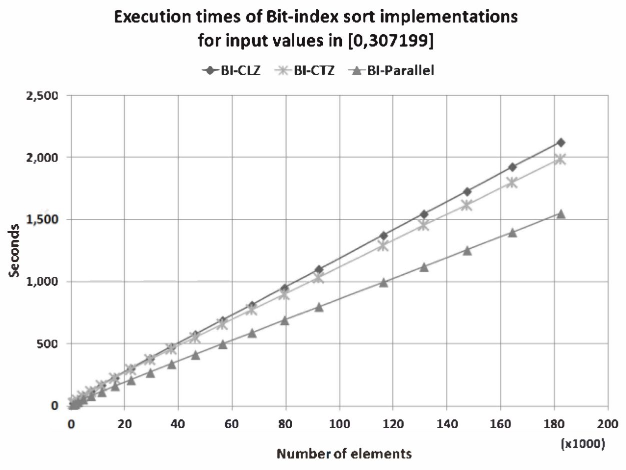

Fig. 7. Execution times of Bit-index sort implementations from values in Table III

V. DISCUSSION

Both Bit-index sort sequential implementations, with *ctz()* and *clz()* functions, show similar execution times when the number of elements in the input sequence is less than 12% of the total range. However, the difference in execution time starts to grow at that point as the number of input elements increases, showing better results the retrieval version using *ctz()*. This trend can be observed in Fig. 7 for input values in interval [0, 307199].

Sorting data for an average case distribution (elements distributed randomly), best case distribution (ordered consecutive elements) and worst case distribution (ordered sequence backwards) show similar execution time as expected from the linear complexity order of Bit-index sort. However, best and worst cases are observed to be slightly faster due to a better spatial locality of cache data on contiguous memory locations.

Results show that Bit-index sort outperforms quicksort increasingly as the number of elements in the input sequences increases. This behaviour was expected due to the linear complexity order of Bit-index sort and the logarithmic complexity order of quicksort. In contrast, when Bit-index sort is compared with counting sort, which is also a linear complexity algorithm, improvement decreases as the number of elements in the input sequence increases. This is caused by the density of bits set in the bit array: when the input sequence contains all the possible values in the range, the amount of operations needed to obtain the ordered output sequence is similar in both algorithms. However, in that extreme case, a slightly better performance in Bit-index sort is expected, since the retrieval stage is implemented using functions with hardware support, and Bit-index sort utilises less amount of additional memory than counting sort to store the counters.

Regarding the parallel version of Bit-index sort, Table IV exhibits speedups for execution times of the Bit-index sort using *ctz()* versus the parallel Bit-index sort using *ctz()* and *clz()* simultaneously, with selected amounts of input data in both data range, [0, 76799] and [0, 307199].

TABLE IV. SPEEDUP LEVELS FOR PARALLEL BIT-INDEX SORT COMPARED WITH BIT-INDEX SORT SEQUENTIAL VERSION USING *ctz()* FUNCTION

Data range [0, 76799]		**Data range [0, 307199]**	
N	***Speedup***	***N***	***Speedup***
50	1.31	500	1.66
250	1.42	2000	1.54
700	1.46	7000	1.49
1600	1.34	16000	1.43
3000	1.36	29000	1.38
6700	1.34	46000	1.33
13100	1.30	67000	1.32
23500	1.28	92000	1.30
42400	1.21	131000	1.30
55000	1.28	164000	1.29

Improvement in execution time varies from around 30% and up to 66% for the parallel Bit-index sort version using two simultaneous threads. Compared with the other sorting algorithm implementations, parallel Bit-index sort displays the best performance overall.

VI. CONCLUSIONS

The sorting algorithm Bit-index sort, presented in this paper, shows a better performance in execution time when is compared with quicksort and counting sort algorithms in the case of sorting elements in a permutation, i.e. an input data sequence without repeated values.

Bit-index sort uses a bit-array, implemented as an array of unsigned integers, to classify input data. Also, Bit-index sort implementations take advantage of hardware support for built-in functions in C compilers in order to obtain a better performance in execution times. From the results obtained, Bit-index sort outperforms quicksort and counting sort when the permutation contains around 70% of the possible input elements, and thus is faster. Improvement in performance increases as the size of range input value and data input increase.

Bit-index sort can be used on input value ranges that include negative integers by adding a shift number in order to map the bit-array index to the input value range. We plan to experiment with an implementation of Bit-index sort that includes repeated values in the input data sequence by associating counters to every bit set.

Parallel implementation of Bit-index sort obtains speedups between 1.3 and 1.6 with two simultaneous threads on the retrieval stage. We also plan to experiment on parallel Bit-index sort implementations for multi-core CPU and GPU increasing the number of simultaneous threads seeking to improve execution performance even further.

REFERENCES

[1] T. Cormen, C. Leiserson, and R. Rivest, Introduction to Algorithms. Cambridge, Mass. U.S.A.: MIT Press, 1990.

[2] C. A. R. Hoare, “Quicksort,” The Computer Journal, vol. 5, no. 1, 1962, pp. 10–16.

[3] S. Carlsson, “Average-case results on heapsort,” BIT Numerical Mathematics, vol. 27, no. 1, 1987, pp. 2–17.

[4] D. Knuth, The Art of Computer Programming V3: Sorting and Searching, Second Ed. Reading Mass. USA: Addison-Wesley, 1998.

[5] I. J. Davis, “A Fast Radix Sort,” The Computer Journal, vol. 35, no. 6, Dec. 1992, pp. 636–642.

[6] R. Merris, Combinatorics, Second Edi. Hoboken, NJ, USA: John Wiley, 2003.

[7] D. Babu and R. Shankar, “Array-indexed sorting algorithm for natural numbers,” in 2011 IEEE 3rd International Conference on Communication Software and Networks, 2011, pp. 606–609.

[8] S. Bao, Y. Xu, D. Zheng, and Z. Zhao, “Novel Sorting Algorithm Based on Guessing Function,” in 2006 IEEE International Symposium on Signal Processing and Information Technology, 2006, pp. 411–414.

[9] Intel Corp., Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2: Instruction Set Reference, Order Numb. 2013, pp. 3–74, 3–76.

[10] IBM Corp., PowerPC User Instruction Set Architecture Book 1 Version 2.02. IBM Corp., 2003, p. 70.

[11] AMD Inc., AMD64 Architecture Programmer’s Manual Volume 3: General Purpose and System Instructions, 24594 Rev. AMD Inc., 2012, pp. 210, 309.

[12] ARM Limited, ARM Developer Suite Version 1.2 Assembler Guide, Release 1. ARM Limited, 2001, pp. 4–38.

[13] Free Software Fundation. Inc., “Using the GNU compiler collection. 5.46 Other built-in functions provided by GCC,” 2005. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Other-Builtins .html. [Accessed: 15-Feb-2013].

[14] Microsoft, “MSDN Library. Visual Studio 2012. Compiler Intrinsics,” __lzcnt16, __lzcnt, __lzcnt64, 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb384809(v=vs.100).aspx. [Accessed: 15-Feb-2013].

[15] Intel Corp., Intel C++ Compiler for Linux. Intrinsics Reference, 312482nd–001 ed. Intel Corp., 2006, p. 21.

[16] NVIDIA Corp., CUDA API Reference Manual, Version 5.0, NVIDIA Corp., October 2012, p. 511.

[17] The Open Group Base, “qsort(),” IEEE Std 1003.1 Issue 6, 2004. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399 /functions/qsort.html. [Accessed: 15-Feb-2013].

[18] OpenMP Architecture Review Board, OpenMP Aplication Program Interface, Version 3. OpenMP Architecture Review Board, 2011, p. 48