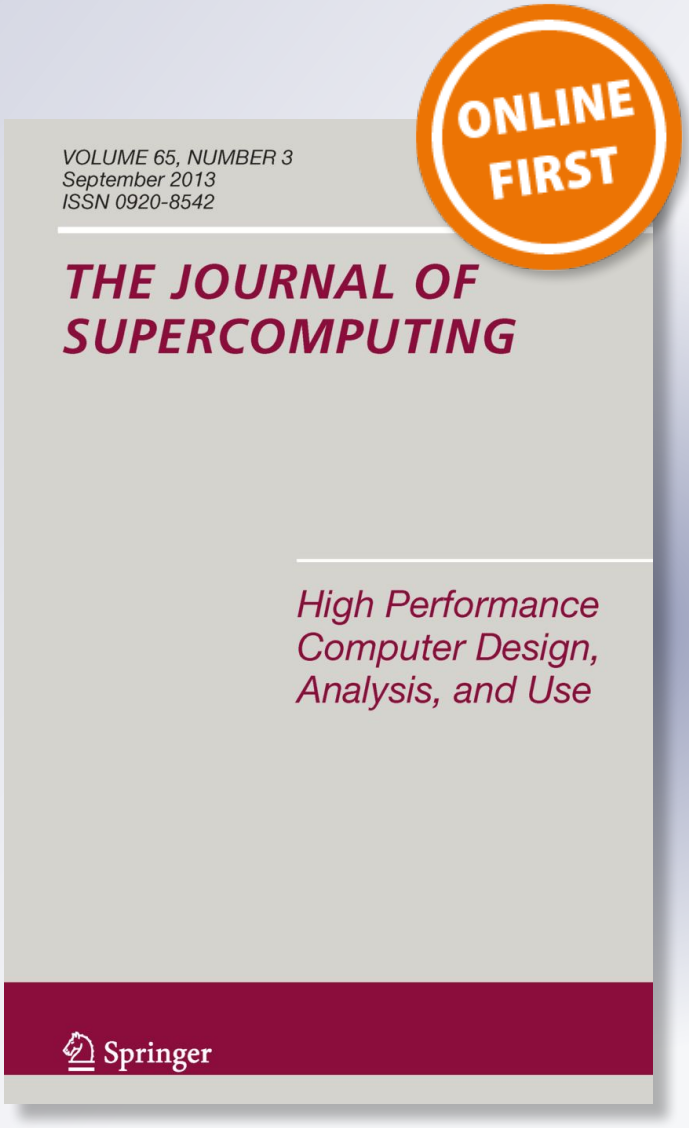
A comparison-free sorting algorithm on CPUs and GPUs

Saleh Abdel-hafeez, Ann Gordon-Ross & Samer Abubaker

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-018-2567-3



Springer

The Journal of Supercomputing
https://doi.org/10.1007/s11227-018-2567-3

A comparison-free sorting algorithm on CPUs and GPUs

Saleh Abdel-hafeez[1] · Ann Gordon-Ross[2,3] · Samer Abubaker[1]

Abstract
This paper presents a new sorting algorithm that sorts input data elements without any comparison operations between the data—comparison-free sorting. Our algorithm's time complexity is on the order of O(N) for both single- and multi-threaded CPU and many-core GPU implementations. Our results show speedups on average of 4.6 × , 4 × , and 3.5 × for single-threaded CPU, 8-threaded CPU, and many-threaded GPU implementations, respectively, for input sizes ranging from 2^7 to 2^{30} elements as compared to common sorting algorithms for a wide variation of element distributions, ranging from all unique elements to a single repeated element. In addition, our proposed algorithm more efficiently utilizes the GPU architecture as compared to a multi-core CPU architecture, showing a speedup of approximately 4 × for input sizes ranging from 2^7 to 2^{30} elements.

Keywords Comparison-free sort · Parallel CUDA code · Multi-core CPUs · Many-threaded GPUs · Big data · Parallel computing · SIMD · Sorting algorithms

1 Introduction

Sorting algorithms have been widely researched for decades [1–5] due to the ubiquitous need for sorting in myriad application domains [6–8]. Subsequently, sorting algorithms have been specialized for particular sorting requirements/situations, such as large computations for processing data [9], high-speed sorting [10], special patterns

✉ Saleh Abdel-hafeez
sabdel@just.edu.jo

Ann Gordon-Ross
anngordonross@ufl.edu

1 Jordan University of Science and Technology, Irbid 22110, Jordan

2 Department of Electrical and Computer Engineering, University of Florida (UF), Gainesville, FL 32611, USA

3 NSF Center for High-Performance Reconfigurable Computing (CHREC), University of Florida (UF), Gainesville, USA

Published online: 30 August 2018

of data types [11], sorting using a single CPU, exploiting the parallelism of multiple CPUs, parallel processing on large grid-computing machines in order to leverage the CPUs' powerful computational capabilities [12], and pooled computing resources for big data processing [13]. Other work has focused on architecting customized hardware designs for sorting algorithms in order to efficiently leverage the hardware resources and provide high-speed hardware processing [14]. However, due to the inherent complexity of sorting algorithms, efficient hardware implementation remains challenging as available computing resources and hardware complexities rapidly increase.

Recent trends in improving sorting performance tailor the algorithms to leverage multi-core CPU and many-core GPU computing resources [15–17], mainly due to the high degree of parallelism provided. Data parallel code is 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 [18, 19]; 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 comparisons within particular input element partitions and further require merging sorted partitions in global memory. Even though new hybrid sorting structures [20, 21] have manifested in recent years to alleviate some of the shortcomings of parallelism, parallel sorting efficiency still has much room for improvement and addressing the outstanding challenges to achieve this efficiency is difficult.

Given the vast variety of sorting options, there is no clear, dominate, generalized sorting algorithm due to many factors [22–25], 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, global and shared memory utilization, the computational phases' execution time similarity, and load balance across parallel resources. As a result, since not all computing domains and sorting algorithms can leverage the high throughput of multi-core CPU and many-core GPU processing, there is a still a great need for novel and transformative sorting methods. In this work, we adapt a hardware-based comparison-free sorting algorithm [12] to parallel software implementations that are capable of utilizing the computing resources of complex CPU- and GPU-based platforms.

Our proposed software-based comparison-free sorting algorithm parallelizes the computations into three forward computational phases: count phase, sum phase, and sort phase, where each phase bundles threads into single-instruction execution units, which is a criteria for hardware utilization optimization. The threads' computations within each phase are arranged in row-major order, which are coalesced into a consolidated access for improved DRAM bandwidth utilization. The threads' execution phases are synchronized using barrier synchronization that coordinates the threads' parallel activities before proceeding to the next phase. Atomic actions between threads are imposed in order to avoid race conditions within each phase. Barrier synchronization and atomic actions are simple and popular methods of coordinating parallel activities with minimal timing overhead [26, 27].

Our proposed sorting algorithm provides end-to-end sorting timing performance that is comparable, and even surpasses, most common parallel sorting algorithms that run on CPU and GPU platforms, to the best of our knowledge. Our algorithm's

performance time complexity on GPU platforms is close to linearly proportional to the number of input elements for input set sizes ranging from 2^7 to 2^{24} elements, giving a computational complexity that is on the order O(N), where N is the number of elements to be sorted, with a small constant scaling factor.

The remainder of this paper is organized as follows. Section 2 discusses relevant related work and highlights goals and challenges, as well as evaluation plans for comparing our proposed algorithm with related work. Section 3 derives the mathematical operation for our proposed comparison-free sorting algorithm with illustrative examples. Section 4 provides a detailed analysis of our single- and multi-threaded CPU implementation along with C-code, and Sect. 5 provides details of our many-threaded GPU implementation with CUDA code. Section 6 presents our simulation results, and Sect. 7 summarizes our contributions and compared with other common sorting algorithms. Finally, Sect. 8 discusses our conclusions and future directions.

2 Related work

In order to provide high-performance operation with a time complexity on the order of O(N), it is critical to develop a sorting method that scales linearly with the number of input elements N and has a small constant factor. In general, single-threaded CPU-based sorting algorithms have moderate performance that scales approximately at O(NLog(N)^2) on average for different data distributions (number of repeated elements and ordering). Recent works [15–21] have developed parallel sorting algorithms that exploit both single-instruction multiple-data (SIMD) instructions and thread-level parallelism. These sorting algorithms reduce the time complexity to a range from O(NLog(N)) to O(Log(N)Log(N)) in the best cases. However, these algorithms' performances are still limited by inefficient operations, such as serial tree comparisons and swapping, unaligned memory accesses, and unbalanced workload across processing threads.

Popular, cutting-edge sorting algorithms harness parallel processing machines, such as many-core GPUs, to maximize performance. However, challenges still exist with respect to effectively utilizing the processing elements with balanced computational workload across all parallel threads. Some new many-core sorting algorithms take advantage of the SIMD instructions and can evenly balance multiple threads' workloads, giving a performance time complexity of approximate O(N); however, this time complexity hides potentially high constant factors that are dependent on the data size, input set size, shared memory access conflicts, and large comparison operations.

In this work, we proposed a comparison-free sorting algorithm that eliminates the use of comparison operations between data elements. Our methodology is based on a sequential, single-threaded processing structure with simple array operations that we extend to parallel computations to effectively harness the computational power of multi-core CPUs and many-core GPUs, resulting in a time complexity of O(N) with a minimal constant factor for large input set sizes and diverse data distributions.

To compare our algorithm's time complexity to related work and common sorting algorithms, we consider commonly used data input set configurations for comparing the performance for different sorting algorithms [31–37]. We evaluate single-threaded

CPU, multi-threaded CPU, and many-threaded GPU implementations for input set sizes ranging from 2^7 to 2^{30} elements for different data distributions including Gaussian, nearly sorted, and reverse ordering for input sets containing only unique elements and varying degrees of repeated elements. We also evaluate varying element values M ranging from values that are much larger than the input set size N ($M > N$) to values of M that are less than or equal to the input set size N ($M \leq N$) [38]. For example, an input set size $N = 4$ elements and cases were $M > N$, $M = N$, and $M < N$; each element can have any arbitrary value regardless of N (e.g., $M = \{1, 1{,}000{,}000, 3, 298\}$), no value larger than N (e.g., $M = \{1, 0, 4, 3\}$), or no value larger than N (e.g., $M = \{1, 0, 0, 3\}$), respectively. These cases enable evaluation of the algorithm's effectiveness in balancing the computational workload across multiple threads, and verification of an O(N) time complexity across diverse architectures and sorting scenarios. To further evaluate the effectiveness in balancing the threads' computational workloads, we also evaluate different percentages of repeated elements in the input set.

3 Mathematical operation

Our comparison-free sorting algorithm minimizes the computational complexity, as well as reducing communication, area, and memory requirements, by eliminating the repetitive comparisons between elements and the data movement between memory and the comparison units. The main operational paradigm of our algorithm is based on array matrix operations, which is suitable and effective for utilizing parallel resources. The input to our algorithm is a K-bit binary bus, which enables sorting $N = 2^K$ input elements. Each element is stored 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 "100,000." For a set of $N = 2^K$ elements, the complete representation contains all binary elements of size one-hot weight $H = N = 2^K$. For example, a $K = 3$-bit input bus can sort/represent $N = 8$ elements, where each element's one-hot weight representation is of size $H =$ 8-bit (i.e., $H = N$).

Even though Sects. 4 and 5 will present additional phases of operation, our algorithm's mathematical principal requires two phases. The initialization phase stores the input elements in an array *IN[.]* of size $N \times 1$, 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 $N \times H$, where each stored element is of size H-bit and $H = N$, giving a transpose memory of size N-bit $\times N$-bit. The evaluate phase effectively sorts the elements by outputting the transposed elements using a matrix multiplication operation between *TM[.][.]* and *IN[.]*, rather than using comparison operations. 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 elements are read from the transpose memory, where each transposed row accesses the element in *IN[.]* and stores the element into the sorted array *SO[.]*, which contains the final sorted data elements. Figure 1 depicts our comparison-free sorting algorithm using matrix multiplication based on linear algebra vector computations and a simple illustrative example. This

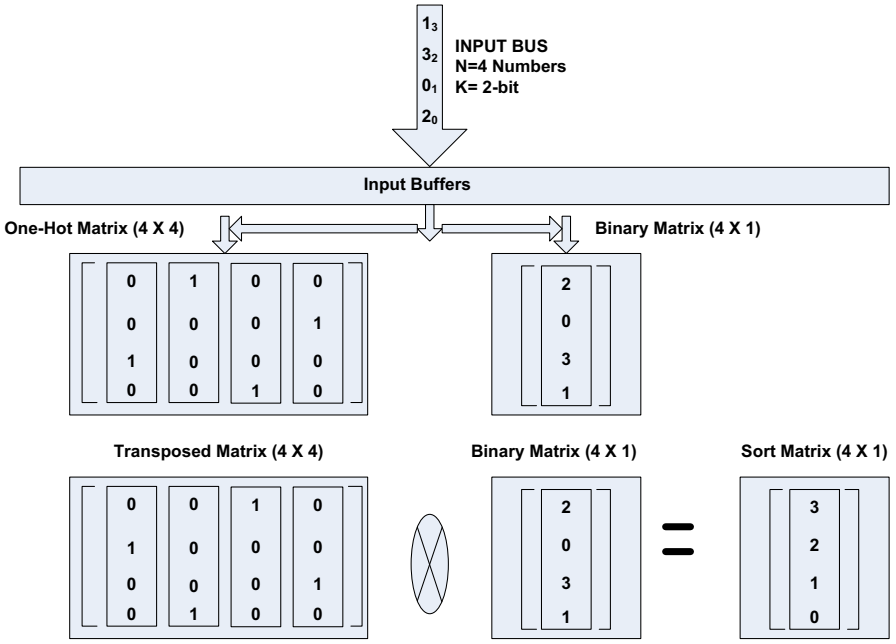


Fig. 1 Comparison-free sorting example using four 2-bit input data elements

example shows our algorithm's functionality for an example with an input set size $N = 4$ four input data elements of size 2-bit, with an initial (random) ordering of {2, 0, 3, 1}, which generates the outputted elements in $S0[.] = \{3, 2, 1, 0\}$.

Duplicated elements are represented using the same vector space, such that the corresponding transpose memory *TM[.][.]* has multiple "1 s" within a column. The number of "1 s" within the column of *TM[.][.]* equals the number of times that element is repeated in the input. The multiple "1 s" accesses the same element in *IN[.]* with no contention/confliction since these elements occupy a different index in *IN[.]*. Consequently, when the column is read, the multiple "1 s" within the associated column is mapped to the same elements in *IN[.]*, which accesses the same element every time that column is read (i.e., the number of times the column that is read is equal to the accumulated number of "1 s" within the entries in the column that is read). For any column with only "0 s" (i.e., the row's associated element is not in the input data), there is no associated read operation and the address pointer into *TM[.][.]* is incremented to the next column.

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 s" except for the last column, which has all "1 s." This case requires $N - 1$ iterations that have no read operations, since all values are "0," plus N iterations for reading the last column that has all "1 s." Thus, the worst-case read operation requires $2N - 1$ iterations.

Considering the sum of the two phases', the best-case sorting time (lower bound) is $2N$ iterations when all N input elements are unique. The worst-case sorting time (upper bound) is $3N - 1$ iterations when all N input 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 of O(N) independent of the data type.

3.1 Mathematical proof

As proof of concept, we present the mathematical proof for our sorting algorithm where all N input elements are unique, which represents the bast-case scenario, and other input element set cases (i.e., different numbers of repeated elements) can be easily derived from this case.

Let

$$L = [a1, \ldots, a\ k] \tag{1}$$

be a given list of k positive integers, and let

$$M = \max[a(1), \ldots, a(k)]. \tag{2}$$

Let $J = J_L$ be the $(k \times M)$ matrix whose entries $J_{r,s}$ are defined by

$$J_{r,s} = \begin{cases} 1, \text{ if } a(r) = s \\ 0, \text{ Otherwise} \end{cases}. \tag{3}$$

Thus, if s does not belong to L (i.e., there is no r such that $a(r) = s$), then the sth column of J will contain all "0 s." If s belongs to L, then the sth column of J will have "1 s" in exactly the locations r where $a(r) = s$.

Supposing that L had no repetitions, let

$$\begin{aligned} LJ &= [a(1), \ldots, a(k)] \\ J &= [b(1), \ldots, b(m)], \end{aligned} \tag{4}$$

which gives

$$b(s) = \begin{cases} s, \text{ if } s \in L \\ 0, \text{ Otherwise} \end{cases}. \tag{5}$$

If $s \notin L$, then all of the values in the sth column of C_s of J are "0 s," and $(s) = L \cdot C_S^T = 0$. If $s \in L$, and if r is the unique value for which $a(r) = s$, then all of the values in the sth column of C_s of J are all "0" except for the value in the rth column, which is "1." Therefore, $() = L\ \ C_S^T = (r) = s$, which proves our claim.

For example, starting with $L = [3, 4, 6]$, then $J = J_L$ would be the matrix

$$J = \begin{Bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{Bmatrix} \tag{6}$$

and

$$LJ = \begin{bmatrix} 0, & 0, & 3, & 4, & 0, & 6 \end{bmatrix}. \tag{7}$$

Let J^* be the matrix obtained by deleting "0" columns from J such that

$$LJ^* = \begin{bmatrix} 3, & 4, & 6 \end{bmatrix}. \tag{8}$$

4 CPU implementations

Our proposed sorting algorithm is based on the mathematical algorithm depicted in Fig. 1. The C-code program has two vector matrices: the input array matrix *IN[.]* of size $N \times 1$ and the outputted sorted array *SO[.]* of size $N \times 1$. The transpose one-hot matrix *TM[.][.]* of size $N \times N$ is substituted for by a count array *CA[.]*, which is realized using a one-dimensional matrix of size $N \times 1$. *CA[.]* stores each binary value that is used as the index into the matrix and the count of the binary value is the matrix's stored value that is associated with that matrix index. For the single- (Sect. 4.1) and multi-threaded (Sect. 4.2) CPU implementations, this structure reduces the storage memory requirements from $N \times N$ to $N \times 1$, making the storage memory efficient and the operations fast when retrieving and storing data.

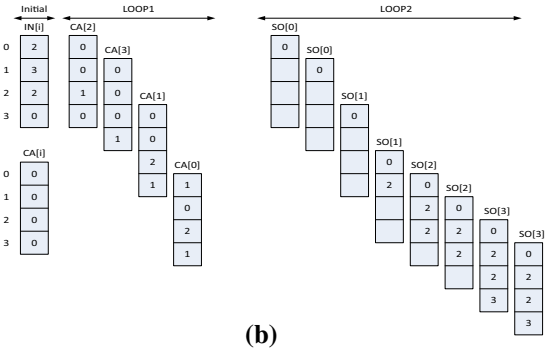
4.1 Single-threaded implementation

The C-code is structured in two phases, the initialization phase and the evaluation phase. The initialization phase is illustrated in the first for-loop in Fig. 2a, where the indices of *CA[.]* record the input elements of size $N \times 1$. The elements in *IN[.]* are read sequentially in the order that they appear in the input sequence (we assume for convenience that *IN[.]* starts indexing at 0). With respect to *IN[.]*, the code exhibits good spatial locality, but poor temporal locality since each element is accessed exactly once. The elements in *CA[.]* are referenced twice during each loop iteration and exhibit good temporal locality with respect to the index vector *CA[.]*. Overall, our 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 memory.

The evaluation phase, which is illustrated in the second for-loop in Fig. 2a, sorts the input elements using *CA[.]* and stores the sorted elements in the sort array vector *SO[.]*. The elements in *CA[.]* and *SO[.]* are read and written sequentially, respectively, resulting in good spatial locality, which affords high performance with small memory requirements and has minor computations. Both loops read/write the elements of the

```
1. Input:  Integer Element IN[0:n - 1 ]
2. Output: Integer Sort SO[0:n - 1 ]
3. count array: char CA[0:n-1] : initialize to zero
4. for i = 0 to n  do
5.         CA[IN[i]] ←CA[IN[i]] + 1
6. endfor
7. Z ←0
8. for i = 0 to n  do
9.         if CA[i]>0  then
10.              SO[Z] ←i
11.              Z←Z+1
12.              CA[i] ←CA[i]-1
13.              i←i-1
14.        endif
15. endfor
```

(a)



(b)

Fig. 2 a Comparison-free sorting C-code for a single-threaded CPU implementation and **b** illustrative example

arrays in row-major order, a characteristic that is more suitable for ANSI-c and the *gcc* compiler as compared to column-major order. Additionally, both loops traverse each element in the arrays sequentially with a reference stride of one (with respect to the element size), which also exhibits good locality [28]. The memory requirements can be further reduced by reusing *IN[.]* for the role of *SO[.]*, which precludes the need for allocating memory for both arrays, as shown in Fig. 2a.

Figure 2b shows an illustrative example of our algorithm for a single-threaded CPU implementation based on the C-code. During the initialize phase, the input sequence array $IN[4] = \{2, 3, 2, 0\}$ and the count array $CA[4] = \{0, 0, 0, 0\}$. Loop 1 (lines 4–6) generates the count array values in four iterations. Loop 2 (lines 8–15) generates the sorted array $SO[4] = \{0, 2, 2, 3\}$ in eight iterations; thus, for every instance of an element in the sorted array, loop 2 adds another extra iteration (i.e., since elements "0" and "3" each occur once in the input set, each of these elements take two iterations in loop 2; since element "1" is not in the input set, this element takes one iteration in loop 2; and since element "2" occurs twice in the input set, this element takes three iterations in loop 2).

4.2 Multi-threaded implementation

Our multi-threaded implementation exploits parallel computing by partitioning our algorithm into several parallel logical flows, where each flow can be assigned to a thread, and each thread operates on a partition (range) of input elements. The concurrent partitions are dictated by the inherent matrix computations and the independent mapping between the one-hot transpose rows in *TM[.][.]* and the input array *IN[.]*. Since context switching and atomic operations require more CPU time for scheduling and swapping data memory, we derive a structure that trade-offs more local memory for less context switching and atomic operation overheads, which improves the sorting performance.

Using the single-thread C-code in Fig. 2a as a basis, we parallelize the first and second loops, as shown in Fig. 3a. Using this structure, each thread has its own weighted counter variable in multiples of the number of threads, and each thread has its own

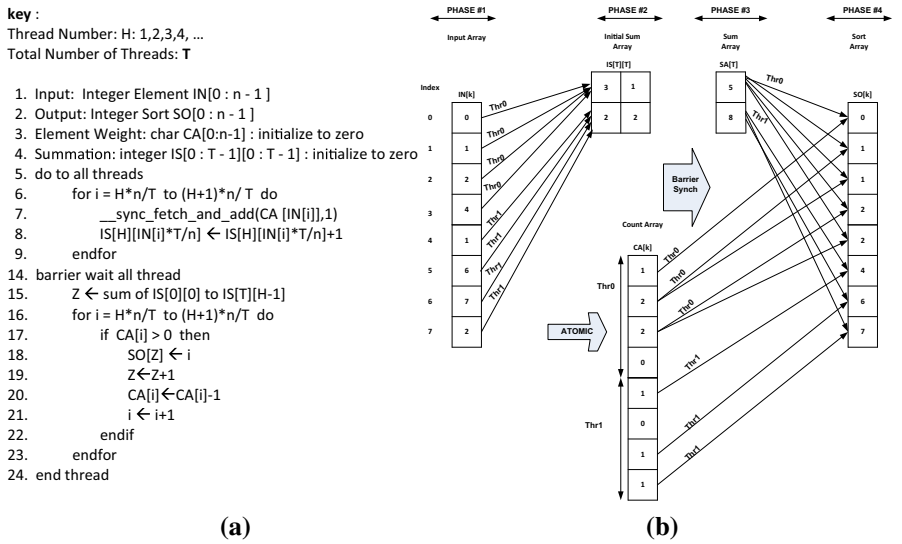


Fig. 3 **a** Comparison-free sorting C-code for multi-threaded CPUs and **b** illustrative example

CT[.] array, which stores the sorting results for that thread's partition of the input set. We insert another loop between the two loops to merge all of the threads' *CT[.]*s to only one *CT[.]* and merge the multiple weight counters in each thread to one-hot weight counter per thread.

Figure 3b shows an illustrative example of our algorithm's phases of operation for a multi-threaded CPU implementation where the number of elements is $N = 8$ and the number of threads is $T = 2$ (Thr0 and Thr1). Initially, the number of elements N in the unsorted array *IN[.]* is evenly divided by the number of threads T (i.e., $N/T = 4$ elements per thread). The first thread operates on the first partition of four elements, the second thread operates on the second partition of four elements, and so on, for larger examples, as clearly demonstrated in the computations in phase #1. Each thread concurrently processes its own partition of the input array *IN[.]* and generates the arrays for phase #2 in parallel.

During phase #2, the initial sum array *IS[.][.]* is generated using the operation on line 8 (Fig. 3a). The count array *CA[.]* is generated using $CA[IN[i]] = CA[IN[i]] + 1$ on line 7, which is an atomic add operation to avoid race conditions between threads accessing the same memory location for repeated elements in *IN[.]*. Barrier synchronization is used after phase #2 (line 14) to ensure that all threads (two threads in this example) have completed phase #2 before proceeding to the next phase. In phase #3, the threads collaborate to generate *SA[.]*. The first element of *SA[.]* is generated by summing the first column of *IS[.][.]*, and the second element of *SA[.]* is generated by summing the second column of *IS[.][.]*. For larger input set sizes, this process would continue for all columns of *IS[.][.]*.

In the last phase, phase #4, the threads collaborate to generate *SO[.]* from *SA[.]* and *CA[.]*. In our two thread example, thread #0 (Thr0) generates the first five elements in *SO[.]*, and thread #1 (Thr1) generates the next three elements of *SO[.]* starting at

index "5" of *SO[.]*. The alignment of the threads (Thr0 and Thr1) is accomplished using *SA[.]* that was generated in phase #3. Each thread reads its partition from *CA[.]* and stores the index number from *CA[.]* in *SO[.]*. If *CA[.]* >0, the value at *CA[.]* is decremented (lines 17 - 22). After all threads finish, the sorted array *SO[.]* is complete.

The memory consumption for our algorithm is space efficient, wherein all arrays are one-dimensional (1D) except for *IS[.][.]*, which is two-dimensional (2D) based on the number of threads ($T \times T$) (e.g., the maximum array size of *IS[.][.]* for an 8-thread example is 64, and the 1D arrays are of size N, the input set size). We point out that there is no further overhead for swapping elements in the arrays nor is there any temporary storage for the merging and comparison operations. Threads independently operate on the data in all computational phases, except for generating *CA[.]* and *IS[.][.]* in phase #2 where multiple threads operate on the same memory location for repeated elements. Memory coherence is maintained using an atomic operation, which is a minor drawback affecting performance.

5 GPU implementation

Our proposed comparison-free sorting algorithm is easily parallelizable due to the algorithm's mathematical nature using matrices and vector algebra, as opposed to sequential parallel tree partitioning and recursive ordering, as is used in most common sorting algorithms [22–25]. Our algorithm's parallelism is inherent since the matrix operations can be streamlined into matrix multiplication operations, where every thread is independently responsible for the row-column product summation computations, a characteristic that is considered to be a key feature for harnessing the processing power of GPUs.

Launching a CUDA kernel creates blocks of threads, where each thread block is partitioned into WARPs. WARP execution is implemented using SIMD hardware, wherein threads are coalesced in a WARP and are executed by a single instruction in the same WARP. Utilizing WARPs in our algorithm helps to reduce hardware computation time and enables some optimizations in servicing memory accesses. Due to the matrix structures' implementations in the source code, minimal code changes are required when changing the number of threads per blocks for sorting different input set sizes.

We propose two methods (method 1 and method 2) for the GPU implementation, which provide high-performance computational solutions that are competitive with recent common GPU-based sorting mechanisms.

5.1 Method 1

Similarly to the mathematical operation outlined in Sect. 3, method 1 is separated into two phases: the initialization phase and the evaluate phase.

5.1.1 Initialization phase

During the initialization phase, the input array *IN[.]* is converted to a one-hot weight representation and is transposed and linearized into a 1D transpose memory *1D TM[.]* of size $N^2 \times 1$. This obviates the need for a 2D memory for efficient memory access; however, the size is still the same, which is large and not memory efficient. The CUDA code:

$$TM[N * IN[i] + i] = 1, \tag{9}$$

where i is the number of threads and *IN[.]* is the input array, which generates a linearized one-hot transposed array from *IN[.]* and incorporates parallel independent thread blocks with the same operation. This single line of CUDA code simplifies three operations, which are the conversion to the one-hot representation, the transpose operation, and the linearize operation, into a single instruction. Only N threads are needed since there are only N elements in the input array *IN[.]*, where each thread operates on N rows of elements in *TM[.]*. All N threads are work independently from each other since every input element has a unique index in *IN[.]*. This holds even in the case where there are more than one element with the same value in *IN[.]*, where each element still needs to have a designated location in *TM[.]*.

Figure 4 shows a detailed example using an input array *IN[.]* of size $N = 4$ elements that is transformed to the transposed one-hot weighted elements in *1D TM[.]* of size $N^2 \times 1$. This example shows how the mapping is constructed between *IN[.]* and *1D TM[.]* assuming that initially the one-hot 2D-weighted matrix *OH[.][.]* is constructed. Even though our CUDA code never instantiates *OH[.][.],* using this assumed structure eases explanation and understanding.

In this example, *1D TM[.]* is an array of size 16×1 and is initialized with "0 s." The first four elements of *1D TM[.]* represent the first column in the assumed *2D OH[.][.]*, the second four elements of *1D TM[.]* represent the second column of the assumed *2D OH[.][.]*, and so on.

5.1.2 Evaluate phase

In this phase, threads are also processed concurrently to sort the elements of the input array *IN[.]* with the aid of *1D TM[.]*, which was created in the initialization phase. For simplicity of explanation, we start by assuming that all of the elements of the input array *IN[.]* are non-repeating (distinct). This results in the assumed *2D OH[.][.]* having a single "1" per column and per row, which presents the mutually exclusive event of coinciding "1 s" between rows and is considered ideal for independent parallel thread multiplications. Simple multiplications of *1D TM[.]* and *IN[.]* compute the sorted array *SO[.]*, a criteria that are performance advantageous for GPU structures. The first four elements of *1D TM[.]* are multiplied and summed with *IN[.]* to generate the first four sorted element in *SO[.]* using the first thread. Concurrently, the second four elements of *1D TM[.]* are multiplied and summed with *IN[.]* to generate the second four sorted elements in *SO[.]* using the second thread, and so on, generating all sorted elements in *SO[.]*. The CUDA code for these operations is:

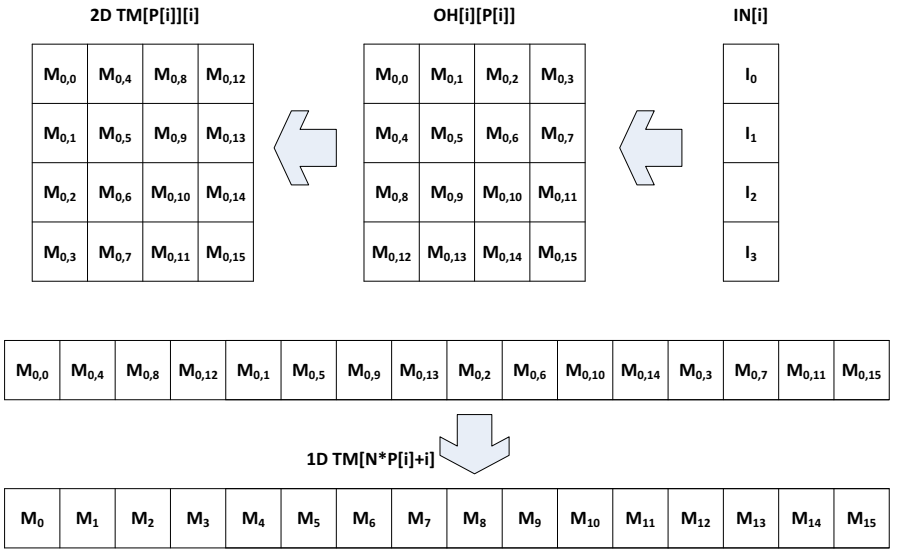


Fig. 4 Example of constructing the 1D transposed one-hot weighted array *TM[.]* from the input array *IN[.]*

$$\begin{aligned} &\text{for } (k{=}0;\ k{<}N;\ k{+}{+}) \\ &\qquad SO[i] \mathrel{+}= TM[i*4+k] * IN[k] \end{aligned} \tag{10}$$

where i is the number of threads and k is the index for the N elements. Thus, each thread is independently responsible for calculating one sorted element in *SO[.]*, and the required number of threads in our example is only four threads since *TM[.]* has only four rows and *IN[.]* has only one column with four elements, as shown in Fig. 3a. However, every thread needs to scan four elements; thus, the four threads require 4^2 computations.

Figure 5a shows a simple *1D TM[.] * IN[.]* multiplication example that introduces the sorted array *SO[.]*. All of these multiplications operate independently of each other, which affords the advantages of parallel thread computations and thus fully harnesses the GPU's computing resources. However, the problem still exists in utilizing memory bandwidth efficiently since every thread needs to scan four elements in *1D TM[.]* concurrently with the other threads, and thus, memory efficiency worsens as the input set size increases.

In general, the input elements can be a mixture of repeating and non-repeating elements; therefore, the proposed multiplication in method 1 [Eq. (10)] must be modified to compensate for the general case. The initialization phase [Eq. (9)] for generating *1D TM[.]* remains the same, while the evaluate phase is modified. The reason for restructuring the evaluate phase is due to the rows in *1D TM[.]* that have multiple "1 s" for repeated input elements; therefore, they are not mutually exclusive in the multiplication operation.

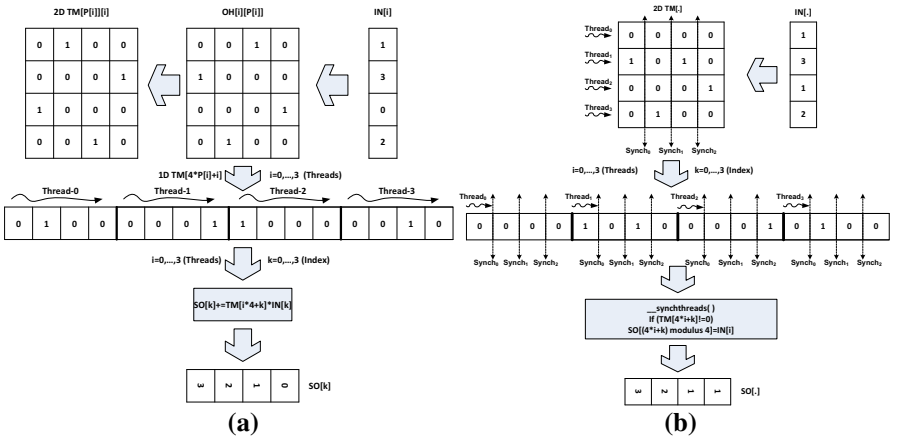


Fig. 5 Detailed example of method 1 for a GPU sorting **a** distinct (non-repeating) elements and **b** arbitrary input elements (repeating and non-repeating)

The restructured evaluate phase is illustrated in Fig. 5b, where threads scan one column at a time in *1D TM[.]*, such that each thread is targeted to one element in the column in *1D TM[.]*. Once all threads have completed computations for the current working column, the threads moved to the next column. The parallel operation of the threads does not proceed to the next column until all threads have completed operation on the current column. This concurrent thread computation behavior for one column at a time is guaranteed using barrier synchronization, which is a feature supported by CUDA. Each thread examines the value in each location of the assigned column in *1D TM[.]*, and if the value is "1," the associated index of that "1" is mapped to the associated index in the input array *IN[.]*, and the sorted array *SO[.]* is updated to store the associated element. The CUDA code for these operations is:

$$
\begin{aligned}
&j{=}0;\\
&\text{for } (k{=}\,0;\, k{<}\,N;\, k{+}{+})\\
&\qquad \_\_synchthreads();\\
&\qquad \text{if } (TM[(i\,{*}N) + k]\,!{=}\,0)\\
&\qquad\qquad SO[j{+}{+}]{=}\,IN[i];
\end{aligned}
\tag{11}
$$

where i is the number of threads and j is the index of *SO[.]*. *__synchthreads()* ensures that the two instructions in the for-loop for the previous iteration have completed for all threads before proceeding to the next iteration. This prevents any thread from scanning the next column of *1D TM[.]* until all threads have completed scanning the current column of *1D TM[.]*. This barrier synchronization between threads is essential for updating and incrementing the sorted array *SO[.]* before proceeding to the next column, as detailed in Fig. 5b.

We note that an atomic operation is no longer needed to protect the j-index from multiple threads incrementing the index per column since every column in *1D TM[.]* has only one element with a "1" even if the element repeats in the input set, as shown

in Fig. 5b. Thus, even though threads compute one column at a time, the mutually exclusive operation in examining the "1" per column is maintained. This way of structuring the threads, as shown in Eq. (11), precludes the use of an atomic operation and thus enhances performance.

The example detailed in Fig. 5b shows *IN[.]* with two distinct elements "2" and "3," and one repeated element "1." Thread-0 (Thr0) is responsible for checking the elements of index 0 to index 3 in *1D TM[.]*, thread-1 (Thr1) is responsible for checking the elements of index 4 to index 7 in *1D [.]*, thread-2 (Thr2) is responsible for checking the elements of index 8 to index 11 in *TM[.]*, and thread-3 (Thr3) is responsible for checking the elements of index 12 to index 15 in *TM[.]*. All threads are synchronized and parallelized for each column computation, and the threads will not proceed to the next column until all threads execute the two instructions in the current iteration denoted by index k.

Clearly, there are several drawbacks to this approach. Since N threads need to scan N elements, a large number of elements must be stored for a long period of time in global memory while waiting to be scanned and operated on by all threads, which does not utilize shared memory bandwidth efficiently since there are many swaps between the global and shared memories. These accesses bottleneck the threads and degrade the overall performance. Global memory is typically DRAM, which is commonly known for slow speed due to inherent hardware technology circuit design factors, such as refresh periods and row buffer misses.

As depicted in Fig. 5b, threads access elements per column, resulting in large data tiling; however, coalescing WARP techniques are not available for this situation since the threads' execution elements are not sequentially available, which leads to poor locality. Therefore, CUDA devices cannot bundle these threads into a single execution instruction and thus cannot attain maximum performance.

5.2 Method 2

Method 2 minimizes each threads' computational load to one element per thread, instead of N elements per thread as in method 1, by emphasizing the key features of our comparison-free sorting algorithm, which sorts without using recursive computations and implies forward-flowing computations. Method 2 organizes thread computations in consecutive and increasing order to compute the elements in parallel using similar timing requirements. Thread blocks are partitioned into WARPs that are restricted to accessing consecutive DRAM locations (i.e., increasing row buffer hits) and operate one control action per thread block, which complements SIMD hardware. Since thread blocks collaboratively load and compute elements in shared memory, rather than accessing global DRAM memory, bandwidth contention is avoided.

Method 2 comprises three phases: the count phase, the sum phase, and the sort phase, which have forward-flowing computation and parallel thread execution as demonstrated in Fig. 5a. If the number of threads is less than the number of elements, the sum phase is split into two phases, as exemplified in Fig. 6a.

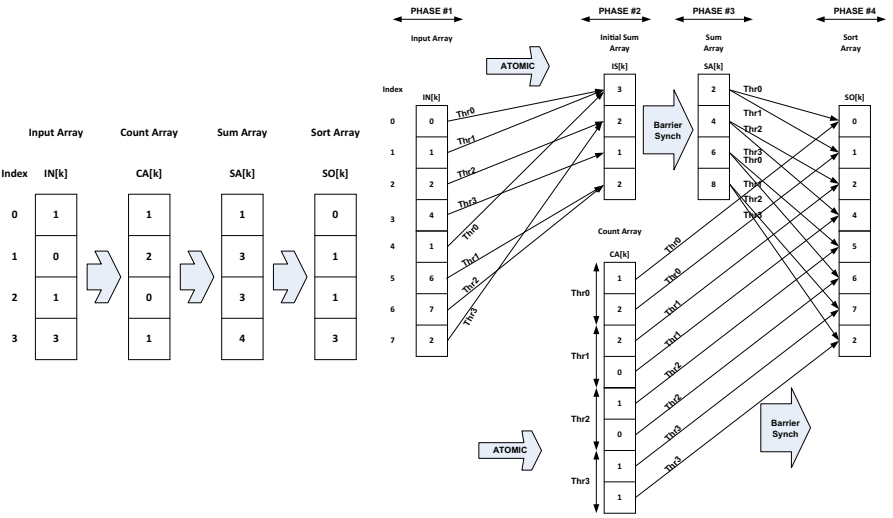


Fig. 6 **a** Detailed example of method 2, and **b** detailed example of method 2 restructured where the number of threads is half the number of elements

5.2.1 Count phase

In the count phase, the number of occurrences of elements in *IN[.]* is counted and recorded in the index of the counter array *CA[.]*. Figure 6a illustrates a simple example for generating *CA[.]* from the input array *IN[.],* where "0" occurs just once, "1" occurs twice, "2" never occurs, and "3" occurs once. The elements' number of occurrences is recoded in *CA[.]* in the elements' associated index. The CUDA code for this operation is:

$$\text{atomicAdd}(\&(CA[IN[i]]),\ 1), \tag{12}$$

where i is the number of threads. In this line of code, four threads execute concurrently since each thread operates on one element in the index of the array *IN[.]*. An atomic addition is needed for the case where more than a single thread is mapped to the same element when there are repeated elements in *IN[.]*. This atomic race condition is considered a computational drawback for repeated elements since all thread instantiated for the repeated elements must wait to exclusively increment the index of *CA[.]*, forcing sequential computation.

The threads are organized such that the threads access *IN[.]* in increasing incremental order, such that thread-0 operates on the element in index *IN[0]*, thread-1 operates on the element in index *IN [1]*, and so on for the remainder N threads. This ordering allows CUDA devices to coalesce thread blocks into WARPs even though all threads in a block can conceptually execute in any order with respect to each other due to the fact that a bundle of threads executes the same instruction and thus requires similar computational workloads.

5.2.2 Sum phase

The sum phase accumulates the count at each index in the sum array *SA[.]* using a parallel prefix-tree adder structure (detailed in [26]). This phase executes several iterations of computations that require $N/2+N/4+N/8+\cdots+1=N-1$ kernel additions. The computational complexity of the parallel prefix-tree adder is O(N), and the CUDA code for this operation is:

```
for (stride = 1; stride <= T/2+1; stride *= 2)
    int index = (blockIdx.x*blockDim.x + threadIdx.x+1)*stride*2 - 1;
    if(index < T+1)
        IS[index] += IS[index-stride]
        __syncthreads();
    for (stride = T/4+1; stride > 0; stride /= 2)
        __syncthreads();
        int index = (blockIdx.x*blockDim.x + threadIdx.x+1)*stride*2-1;
        if(index+stride < T+1)
            IS[index + stride] += IS[index];
```

(13)

where T is the number of threads. The if statements check for a half-partition of elements away from each other, rather than checking every neighboring element for addition operations, which enables a thread convergent operation by grouping the threads into WARPs such that each WARP is either a half-partition away or not (i.e., each WARP will or will not execute the addition operation). This structure is contrary to previous methods, where each even thread index executes the addition operation or not. We refer the reader to [26, 27] for additional details.

5.2.3 Sort phase

After determining the accumulated sum and count values for each index for the input array *IN[.]*, the sorted array *SO[.]* can be evaluated for each index using parallel thread computations. The sum values along with the count values, which were recorded in *SA[.]* and *CA[.]*, respectively, pinpoint the start and end indices in *SO[.]*, respectively. As shown in Fig. 6a, thread-0 computes *SO[0]*, which starts at index 0 and ends at 0 due to $CA[0]=1$, thread-1 computes *SO[1]*, which starts at index $SO[0]=1$ and ends at index 2 since $CA[1]=2$, and thread-1 also computes *SO[2]*. We note that thread-2 does not compute *SO[2]* since $CA[2]=0$. Finally, thread-3 computes *SO[3]*, which starts at $SA[2]=3$ and ends at index 3 since $CA[3]=1$. Using this operation, threads no longer need to wait on the other threads to determine their start and end indices in the sorted array. The CUDA code for this operation is:

```
for (; i2<(i+1)*(N/(T*blk)); i2++)
    if (CA[i2]>0)
        IN[SO[i]++] = i2;
        CA[i2]--;
        i2--;
```
(14)

Atomic operations are no longer needed since each thread examines a unique section of memory. For the case of repeated elements, the count value in *CO[.]* presents a race condition in memory, as is the case in the above example, where thread-1 and thread-2 both try to compute *SO[2]*; however, the count value associated with thread-2 is 0, while the count value associated with thread-1 is 1.

We can observe from the example in Fig. 6a that the threads' workloads are not uniform. Thread-1 computes two values in *SO[.],* which are *SO[1]* and *SO[2]*, thread-0 computes one value, which is *SO[0]*, thread-2 computes no value in *SO[.]*, and thread-3 computes one value, which is *SO[3]*. This non-uniformity in the last phase reduces locality, which worsens as the input set size increases and memory is not able to cover neighboring threads. In this case, threads are not coalesced in an efficient way for WARP hardware control grouping, which reduces SIMD performance.

Since computations on large input sets suffer from there being only a limited number of thread blocks available, we restructure Fig. 6a, b by introducing the initial sum array *IS[.]*. The example in Fig. 6b shows a sample case where there are eight input data elements (i.e., *IN[.]* of size 8×1) and the number of available processing threads is half of the input set size (i.e., there are four threads). *CA[.]* is of size 8×1 and is generated by the four threads running in parallel, such that the first four elements in *CA[.]* are computed by the four running threads. After the four threads evaluate the four elements in *CA[.]*, the threads proceed to operate on the second four elements in *CA[.]*. Barrier synchronization is used to evaluate the two bundles of four elements in *CA[.]*. The CUDA code for these operations is:

```
while (i1 < size)
    atomicAdd(&(CA[IN[i1]]), 1);
    atomicAdd(&IS[(IN[i1]/(kal))+1],1);
    i1 += stride;
```
(15)

Each element in *IS[.]* is computed by processing two elements in *IN[.]*, such that the four threads compute the first two elements in *IS[.]*. Once the threads complete, the threads compute the second two elements in *IS[.]* using barrier synchronization. The basic operation for evaluating the elements in *IS[.]* is demonstrated as follows. Thread-0 checks the value in $IN[0] = 0$, divides the value by "2," which results in "0," increments that result to "1," which is subsequently stored in $IS[0] = 1$. Concurrently and atomically, thread-1 checks the value of $IN[1] = 1$, divides the value by "2," which results in "0," increments that result to "2," which is subsequently stored in $IS[0] = 2$. Concurrently and atomically, thread-2 checks the value of $IN[2] = 2$, divides the value by "2," which results in "1," increments that result to "1," which is subsequently stored in $IS[1] = 1$. Concurrently and atomically, thread-3 checks the value of *IN[3]*

$=4$, divides the value by "2," which results in "2," increments that result to "1," which is subsequently stored in $IS[2]=1$. After these operations complete, the four threads similarly compute the second two elements in *IS[.]* using the second four elements of *IN[.]*, and so on until all elements have been processed.

The parallel adder sum array *SA[.]* in Fig. 6b is generated similarly as in Fig. 6a by having all threads accumulating the addition values pointed to by the index in *IS[.]*. We note that the size of *IS[.]* is only 4×1, and thus, the size of *SA[.]* is also 4×1, which is equal to the number of threads. Once all threads compute *SA[.]*, the threads start computing the sorted array *SO[.]* in parallel based on the values in *CA[.]* and *SA[.]*, as shown in Fig. 6b.

Figure 7 summarizes the complete CUDA code of our comparison-free sorting algorithm GPU implementation. The design leverages parallelism in the working threads, which forms the operational structure and the phases of computations using barrier synchronization between the phases. The first and last phases of computation require the longest computation time. In the first phase, the repeated elements force the threads to have a more diverse mixture of sequential and parallel behavior. In the last phase, the repeated elements force the threads to have an unbalanced utilization of the WARPs, which reduces memory usage efficiency. Based on these, we can estimate the asymptotic boundaries of the computational time complexities by using a valid assumption that the best-case, lower-bound computational time occurs when all elements in the input set are distinct, and the worst-case, upper-bound computational time occurs when all elements in the input set are a single repeated element.

The overall computational time for method 2 is near-linear since N threads operate on N elements, contrary to method 1 where N threads operate on N^2 elements. Additionally, method 2 uses no backward-flowing computations that require storing data for comparisons in global memory for a long period of time. Finally, the majority of threads in method 2 benefit from locality in accessing and operating on neighboring data, which is more efficient for memory accesses.

Memory coherence in GPUs has a minor effect since constant memory variables have an interesting impact on the caches in massively parallel processors. The GPU's hardware can aggressively cache the constant variables in level one cache, and the design of the caches in these processors is typically optimized to broadcast a value to a large number of threads. As a result, when all threads in a WARP access the same constant memory variable, the memory provides a tremendous amount of bandwidth to satisfy the threads' data requirements. Therefore, the threads benefit from spatial locality in order to coalesce these threads into a single WARP and reduce coherency effects. In our illustrative example in Fig. 6b, threads benefit from spatial locality in all phases, except the last phase, where threads are not evenly divided for generating *SO[.]* for the case of repeated elements. In Sect. 6, we evaluate several distributions of repeated elements in the input set, including all unique elements (repeated 0%), only a single repeated element (repeated 100%), and a trade-off point where elements are semi-repeated (repeated 70%). Results show that repeated 70% has nearly identical performance as all unique elements, and repeated 100% only slightly reduces the performance, but we point out that repeated 100% is not a realistic input set for sorting applications.

```
1. Input: Integer Element IN [0 : n - 1 ]
2. Output: Integer Sort SO [0 : n - 1 ]
3. Count array: Integer CA[0:n-1] ← initialize to zero
4. Initial Sum: Integer IS[0 : T* blk+1] ← initialize to zero
5. do to all threads
6.          for i =blockIdx.x*blockDim.x + threadIdx.x   to  n   do
7.                              atomicAdd(&(CA[IN[i]]), 1)
8.                              atomicAdd(&IS[(IN[i]/( n/(T*blk)))+1],1)
9.                              i ← i+blockDim.x*gridDim.x
10.         endfor
11. __syncthreads()
12.         for stride = 1 to  ((T*blk)/2+1)     do
13.                               index ← (blockIdx.x*blockDim.x + threadIdx.x+1)*stride*2 -1
14.                               if(index<(T*blk+1)) then
15.                                       IS[index] ← IS[index] + IS[index-stride]
16.                               endif
17.                               __syncthreads()
18.                               stride ← stride*2
19.       endfor
20.    for stride =( T*plk)/4+1 to  0
21.                              __syncthreads()
22.                              index ← (blockIdx.x*blockDim.x + threadIdx.x+1)*stride*2 – 1
23.                              if(index + stride< (T*blk )+1) then
24.                                       IS[index+stride] ← IS[index + stride] + IS[index]
25.                              endif
26.       endfor
27. __syncthreads()
28.       i1 ← blockIdx.x*blockDim.x + threadIdx.x
29.       for i=(blockIdx.x*blockDim.x + threadIdx.x)* (n/(T*blk))   to
          ((blockIdx.x*blockDim.x + threadIdx.x)+1)* (n/(T*blk))  do
30.                               if (CA[i] >0) then
31.                                       SO[IS[i1]]  ← i
32.                                       IS[i1] ← IS[i1] +1
33.                                       CA[i] ← CA[i]-1
34.                                       i ← i-1
35.                               endif
36.       endfor
37. end thread
key :
Number of Thread in block : T
Number of block :blk
```

Fig. 7 CUDA code for our comparison-free sorting algorithm for method 2 on GPUs

6 Simulation results

In this section, we evaluate the performance of our proposed comparison-free sorting algorithm using increasing input set sizes in powers of 2. We sort integer input data for different data input set distributions, including random distribution, reverse ordering, nearly sorted, Gaussian distribution, for different percentages of repeated elements [i.e., 70% of the elements are repeated (repeated 70%) or 100% of the elements are repeated (repeated 100%)] and entirely unique data (repeated 0%). These distributions are frequently used to evaluate sorting algorithms and reveal execution time variations and sensitivity to different sorting scenarios. We perform simulations using single- and

Table 1 Experimental CPU characteristics

CPU system specifications	Intel CoreTM I7-3770
Clock speed	3.4 GHz
Number of cores × number of threads	4 × 8
Hyperthreading	Yes
L1/L2 (KB)/socket	32/256
RAM size	8 GB
RAM bus	DDR3-1333/1600
Misc	Smart Cache 8 MB, DMI 5GT/s

Table 2 Experimental GPU characteristics

GPU system specifications	NVIDIA TESLA K20M
Number of CUDA parallel processing cores	2496
Peak teraflops	1 TFLOP
Clock speed	706 MHz
Interface	320-bit
Memory size	5 GB
Graphics bus	PCI express 2.0 × 16

multi-threaded CPU implementations with the architectural characteristics in Table 1. Even though this CPU architecture can support multi-threading, we instantiate only one thread for the single-threaded experiments (Sect. 6.1) and instantiate 4 and 8 threads for the multi-threaded experiments (Sect. 6.2) executing the C-code in Sect. 4.1 and derived in Fig. 2 and the C-code in Sect. 4.2 and derived in Fig. 3, respectively. Table 2 depicts the architectural characteristics for our GPU implementation with 16,382 threads (16 threads per block) (Sect. 6.3) executing the CUDA code in Sect. 5 and derived in Fig. 7. Finally, to evaluate thread workload balancing, we simulate cases where the elements' values M can be much larger than the input set size N (Sect. 6.4). We report the actual sorting execution time in seconds and take into account all memory copies and contention, as well as context switching times.

6.1 Single-threaded CPU

Figure 8a depicts the logarithmic scale of execution time in seconds for our algorithm using a single-threaded CPU for varying input data set sizes and different input data distributions and repeated element occurrences. These results shows that the computation times for these sorting scenario variations are almost the same for input sizes below 2^{24} elements, which shows that within this range, the algorithm's computation time is insensitive to and non-biased toward different sorting scenarios. Other sorting algorithms can show large computation time variations for different sorting scenarios [29, 30] for any input set size.

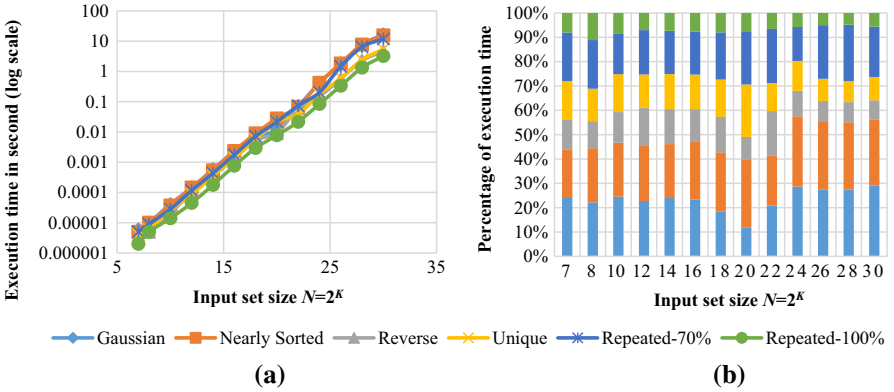


Fig. 8 **a** Logarithmic execution time in seconds and **b** percentage of execution time for our comparison-free sorting algorithm for varying input sizes and different sorting scenarios (different data distributions and number of repeated elements) on a single-threaded CPU

Figure 8b further evaluates these findings by illustrating the percentage of execution time for sorting scenarios. The execution time variations begin to manifest for input set sizes greater than 2^{24} elements, at which point Gaussian distribution and repeated 70% exhibit the worst execution times, and repeated 100% and all unique elements exhibit the best execution times since these sorting scenarios have the greatest memory locality. These execution time variations dictate the upper and lower asymptotic bounds on the sorting time. The average ratio between the best and worst execution time is $1.3\times$ for input set sizes ranging from 2^8 to 2^{30} elements. As compared to other sorting algorithms, quicksort has an average variation ratio of N^2 between the best and worst sorting times based on different sorting scenarios [29, 30].

6.2 4-/8-threaded CPU

Figure 9a shows the logarithmic execution time in seconds for our 4-/8-threaded CPU implementations compared to the single-threaded (1-threaded) CPU implementation for varying of input set sizes with Gaussian distribution. The single-threaded CPU outperforms the 4-/8-threaded CPU for small input set sizes less than 2^{12} elements, with an average improvement of $7.4\times$. Figure 9b depicts more detailed results with execution time in seconds for 1-/4-/8-threaded CPUs. Figure 10 shows the percentage breakdown of execution time for 8-threaded CPU. These results show that the 4-/8-threaded CPUs outperform the single-threaded CPU for input set sizes greater than 2^{20} elements, showing the effectiveness of parallelism for large input set sizes, with an average improvement of $1.8\times$ over a single-threaded CPU.

Figure 11 shows the memory consumption in bytes for different input set sizes with Gaussian distribution. The results show memory efficiency, requiring less than 1 GB for sorting input set sizes less than 2^{26} elements. The memory consumption becomes exponential for input set sizes greater than 2^{27} elements and reaches about 5 GB for input set sizes of 2^{30} elements; however, we point out that this memory requirement is

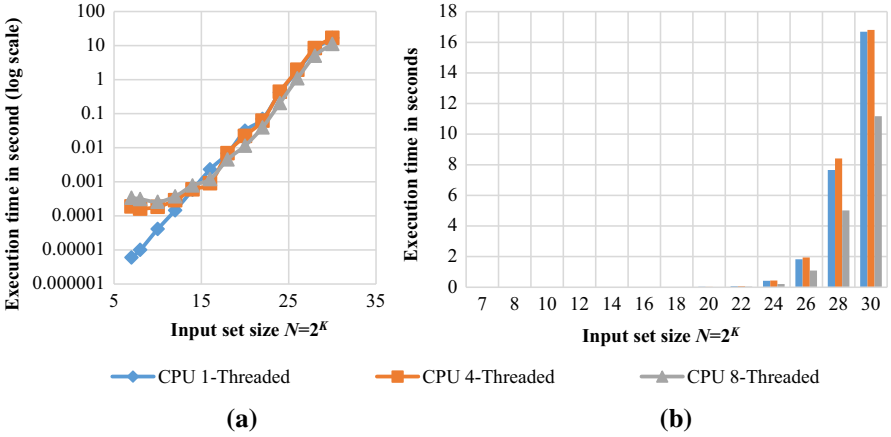


Fig. 9 **a** Logarithmic execution time in seconds and **b** execution time in seconds for our comparison-free sorting algorithm for varying input sizes and Gaussian distribution for 1-/4-/8-threaded CPUs

Fig. 10 Percentage of execution time for our comparison-free sorting algorithm for varying input sizes and Gaussian distribution for 1-/4-/8-threaded CPUs

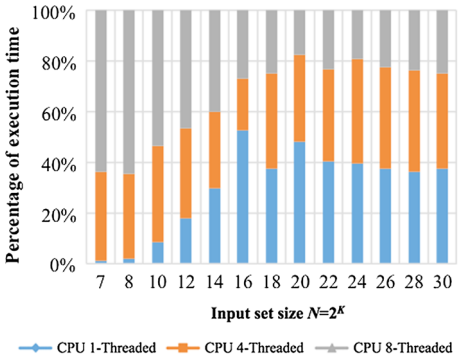
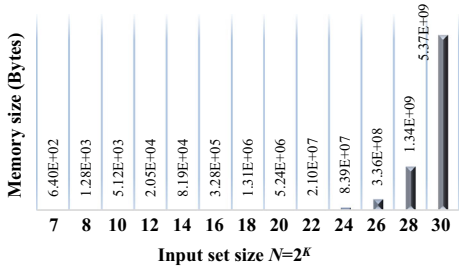


Fig. 11 Memory consumption for our comparison-free sorting algorithm with respect to data input set size for an 8-thread CPU



still competitive as compared to common sorting algorithms [27–30] that are on order of 10 GBs for processing similar input set sizes ranging from 2^{27} to 2^{30}.

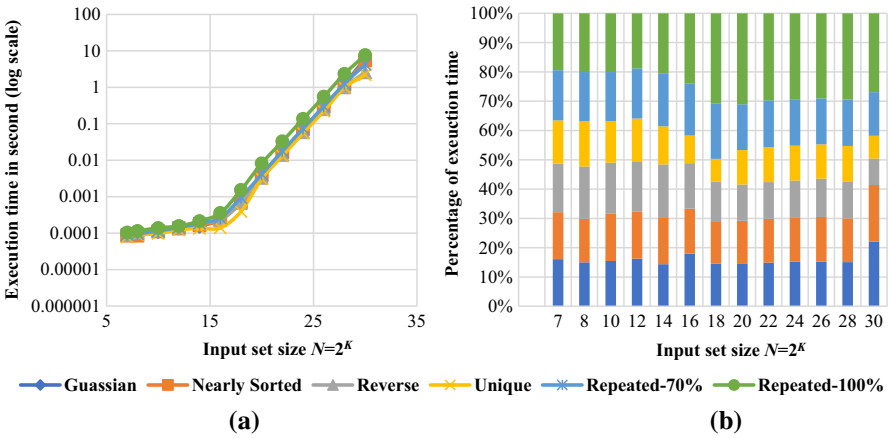


Fig. 12 **a** Logarithmic execution time in seconds and **b** percentage of execution time for our comparison-free sorting algorithm for varying input sizes on a GPU

6.3 16,384-threaded GPU (16 threads per block)

Figure 12a shows the logarithmic execution time in seconds for varying input set sizes and different sorting scenarios. These results reveal the asymptotic execution time boundaries for different ranges of input set sizes. The worst-case computing time is when the input set has a high percentage of repeated elements, as is the case of a single repeated element (repeated 100%). This case is predicted in Sect. 5.2 to worsen the computation time of the first and last phases; thus, this sorting scenario dictates the worst-case, upper-bound asymptotic execution time. The best-case, lower-bound asymptotic execution time occurs when all of the input elements are unique, as also predicted in Sect. 5.2.

Since more realistic data types have a Gaussian distribution or only a few unique elements, Fig. 12b evaluates the percentage of execution time required for different sorting scenarios. These results show that more realistic sorting scenarios exhibit computing times close to the lower-bound asymptotic curve and that the variation in execution time for different input set sizes is small for the GPU, which is approximately $1.1\times$ for a wide range of input set sizes.

Figure 13a shows the logarithmic execution time in seconds for the 1-/8-threaded CPUs and GPU for varying input set sizes with Gaussian distribution. The single-threaded CPU has the best performance, with an average performance increase of approximately $6.7\times$ and $4.2\times$ as compared to the 8-threaded CPU and the GPU, respectively, for input set sizes smaller than 2^{13} elements. However, for input set sizes greater than 2^{13} elements, the GPU's performance surpasses both CPUs by an average of $5.3\times$.

Figure 13b shows the percentage of execution time for the 1-/8-threaded CPUs and the GPU for varying input set sizes with Gaussian distribution. These results analyze the conclusions drawn in Fig. 13a and show that the GPU outperforms the 8-threaded CPU for any input set size and that the computing performance for the GPU has

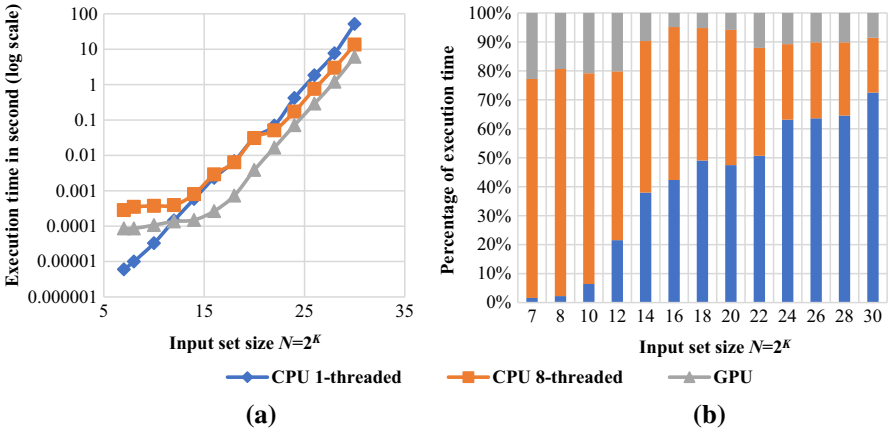


Fig. 13 **a** Logarithmic execution time in seconds and **b** percentage of execution time for our comparison-free sorting algorithm for varying input set sizes with a Gaussian distribution for 1-/8- threaded CPUs and a GPU

the tendency to be closer to linear rather than exponential for large input set sizes greater 2^{20} elements. On the contrary, other common sorting algorithms usually grow exponentially in performance beyond 2^{20} elements or have a large scalar factor on the order of 100 or more for linear scale behavior [34].

Figure 14 shows the logarithmic execution time in seconds for our comparison-free sorting algorithm's different phase's of execution, the count, sum, and sort phases, for the CPU, which were derived in Fig. 6 and Sect. 5.2. These results show that the worst-case computation time is dictated by the sort phase for large input set sizes above 2^{20} elements. The sum phase, as predicted in Sect. 5.2, has the lowest computation time for all input set sizes, while the count and sort phases have larger computation times, as illustrated in Sect. 5.2. We note that the sort phase has the worst computation time due to repeated elements that force unbalanced thread workload. This computational bias is a result of the WARPs effectively utilizing the control hardware resources for parallel threads performing SIMD computations.

Figure 15 shows memory consumption in bytes for varying input set sizes for our comparison-free sorting algorithm for the GPU. These results have an efficient constant factor for input set sizes ranging from 2^{14} to 2^{28} elements, which shows efficient SIMD operation. The maximum global memory required for the maximum input set size of 2^{28} elements is on the order of 1 GB. We note that the memory consumption can be reduced by using the latest version of CUDA, which allows using character-type constants instead of integer-type constants as in conventional CUDA.

6.4 CPU and GPU simulations for $M > N$

In this section, we evaluate sorting scenarios where the largest element's value M can be greater than the input set size N to garner more insight on the computational complexity behaviors of our proposed algorithm. For example, considering an input

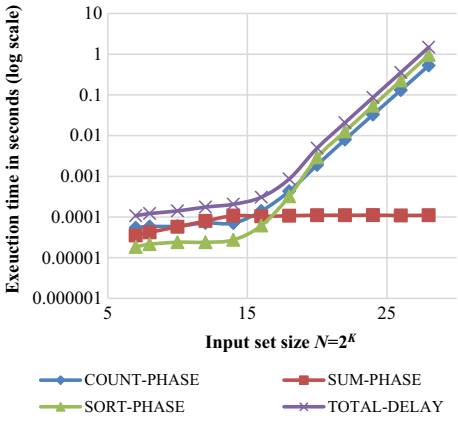


Fig. 14 Logarithmic execution time in seconds for our comparison-free sorting algorithm's different computational phases for varying input set sizes with Gaussian distribution for the GPU

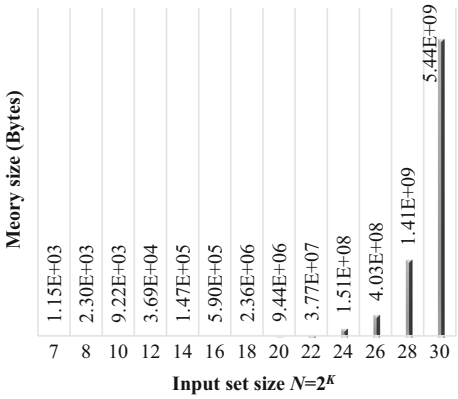


Fig. 15 Memory consumption for our comparison-free sorting algorithm for varying input set sizes the GPU

set size $N = 1024$ elements, and where the values of these elements can range from "0" to "1000 * N," $M = 1{,}024{,}000$. This maximum data value with respect to the number of elements provides a 99.9% increase, which is considered sufficient to draw over-arching conclusions about the execution time performance and the behavior of our algorithm.

For these experiments, we assume the following constraints from our prior simulations: the CPU and GPU use the characteristics depicted in Tables 1 and 2, respectively; the input set size N ranges from 2^7 to 2^{30} elements with Gaussian distribution; the data values M range from "0" to "1000 * N."

Figure 16 shows the summary of (a) the logarithmic execution time in seconds and (b) percentage of execution time for input set sizes N where the elements' values M range from 0 to 1000 * N. These results show that the many-threaded GPU's performance is not influenced by the change in M regardless of how M relates to N (i.e., $M < N$ or $M > N$) as is evident with comparisons where $M < N$ in Figs. 12 and 13; thus, the degree of execution time is somewhat linearized toward the order of O(N). We also note that the CPU performance on 1-/8-threaded CPUs has slightly deviated performance for small input set sizes, but for large $N > 2^{24}$ elements, the performance

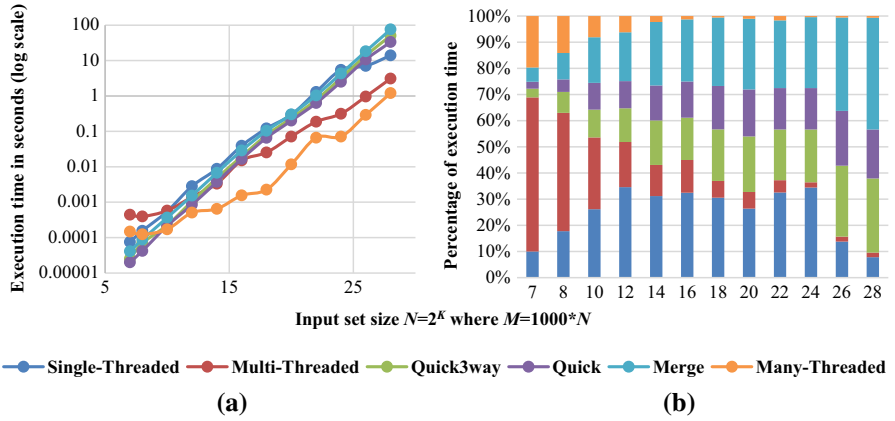


Fig. 16 **a** Logarithmic execution time in seconds and **b** percentage of execution time for our comparison-free sorting algorithm for varying input set sizes and corresponding element value ranges with Gaussian distribution for single- and multi-threaded CPUs and a many-threaded GPU as compared to common sorting algorithms

is almost the same and there is no drawback in execution time, wherein the results are almost identical to that reported in Figs. 8, 9, 10, 11, 12 and 13.

7 Summary and comparison with prior results

Table 3 summarizes the key features of our comparison-free sorting algorithm for 1-/8-threaded CPUs and a many-threaded GPU. Row 1 compares our sorting algorithm's performance increase as compared to common sorting algorithms running on similar platforms. Row 2 compares the performance increase for the GPU as compared to the CPUs. Row 3 shows the memory consumption for all implementations, which is constant at approximately 5 GB. We note that the GPU provides superior performance as compared to the CPUs with the same memory usage. Rows 4 and 5 show the sorting scenarios that give the best and worst performance, respectively, for each implementation, and row 6 shows the variation ratio in the performance between the best and worst performance. These results show that the GPU's performance is least effected by the sorting scenario and only a minor effect on the CPUs performance when the input size is larger than 2^{24} elements. Rows 7 and 8 show the input set size ranges that show strict linear or logarithmic growth rates, respectively, which shows that the GPU has an even larger linear growth rate range as compared to the CPUs; however, we note that all implementations actually scale linearly for all data sizes with an execution time complexity near $O(N)$ with small constant factor. Row 9 emphasizes this fact, showing the logarithmic expression rate, which is a linear expression of rate < 0.5 and DC offset < 0.5. In these expressions, the GPU has the highest performance with a rate of change against input set sizes of powers of 2 with the lowest DC offset close to zero. The last row shows the number of lines of code required for each implementation for non-commercial use, which shows our algorithm's simplicity.

Table 3 Our proposed comparison-free sorting algorithm's key features for a single- and multi-threaded CPU and a many-threaded GPU

Algorithm features	Implementation		
	CPU 1-threaded	CPU 8-threaded	GPU many-threaded
1. Performance increase compared to related sorting algorithms	4.6 × –6 ×	4 × –23 ×	3.5 × –100 ×
2. GPU performance increase over 1-/8-threaded CPU	5.6 ×	4.1 ×	N/A
3. Memory consumption $N = 2^{30}$	≈5 GB	≈5 GB	≈5 GB
4. Best performance data type	Repeated 100%	Unique	Unique
5. Worst performance data type	Repeated 70%	Repeated 100%	Repeated 100%
6. Variation ratio between best and worst performance	1.3 ×	1.3 ×	1.1 ×
7. Linear growth rate range	$<2^{21}$	$<2^{23}$	$<2^{26}$
8. Logarithmic growth rate range	$<2^{30}$	$<2^{30}$	$<2^{30}$
9. Log interpolation equation versus data 2N	$0.6*N*\log(e)+\log(0.4)$	$0.55*N*\log(e)+\log(0.5)$	$0.5*N*\log(e)+\log(0.9)$
10. Code size in number of lines	15	24	37

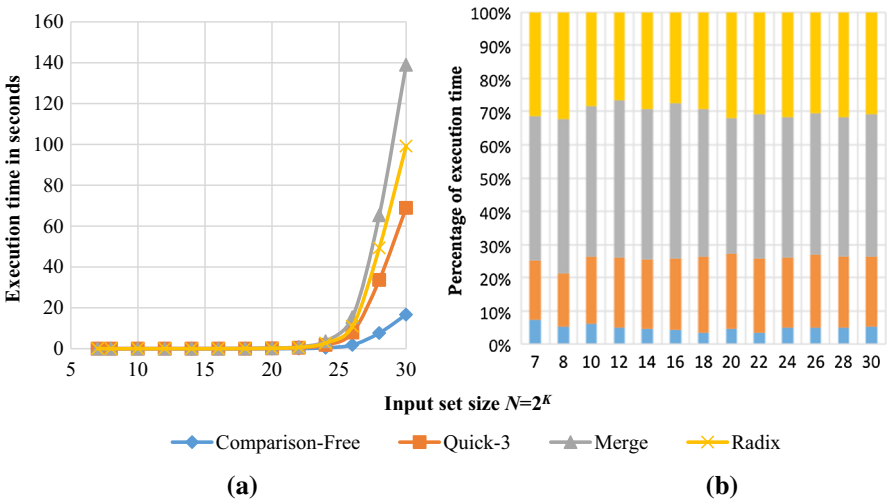


Fig. 17 a Execution time in seconds and **b** percentage of execution time for our comparison-free sorting algorithm as compared to other common sorting algorithms for varying input set sizes with Gaussian distribution on a single-threaded CPU

Figure 17a, b compares our algorithm's single-threaded CPU implementation's execution time in seconds and percentage of execution time, respectively, with other common sorting algorithms [29, 30] for varying input set sizes with Gaussian distribution. These results show execution time reductions for our sorting algorithm of approximately 4.6 × as compared to quicksort and 6 × as compared to merge sort and radix sort. Based on these results, to the best of our knowledge, our algorithm can be considered as one of the fastest sorting algorithms as compared to other common sorting algorithms.

Table 4 compares the execution time in seconds for our algorithm for an 8-threaded CPU compared to other common sorting algorithms executed on a quad-core CPU for large input sizes. These results show the performance advantage of our algorithm, which is mainly due to the vector thread operations and forward-flowing computations as compared to other sorting algorithms that require several backward-flowing computations and dependent thread operations. Our sorting algorithm also obviates the need for a large memory by only operating on smaller partitions of the input set and avoids using the transposed one-hot memory as compared to other sorting algorithms that use large memories for backend merge sorting computations and backward-flowing comparisons between data elements. Table 5 shows similar performance advantages for the GPU implementation compared to other common GPU sorting algorithms [35–37].

Figure 18 shows the memory consumption in bytes for our sorting algorithm as compared to other common sorting algorithms for varying input set sizes with Gaussian distribution. The results show that for input set sizes less than 2^{25} elements, our algorithm's memory consumption is comparable to other sorting algorithms and shows efficient improvements as the input set size increases, and outperforms all other sorting

Table 4 Sorting time in seconds for common sorting algorithms parallelized on a quad-core CPU and our proposed sorting algorithm

Algorithm	Input set size				
	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}
Butterfly sort [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 sorting algorithm with an 8-threaded CPU	0.017	0.058	0.234	1.08	4.41

Table 5 Sorting time in second for common sorting algorithms parallelized on a GPU and our proposed sorting algorithm

Algorithm	Input set size					
	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	GPU device
Merge [35]	0.024	0.13	0.41	1.68	2.64	GTX 280 (30 SMs)
Quicksort [36]	22	62.7	113	228	328	GTX 8800 (512MIB)
Rank [37]	1.7	21.5	92.2	156	285	Quadro 6000
Odd–even [38]	0.83	7.4	36.7	92	143	Quadro 6000
Bitonic [39]	0.054	0.63	1.3	1.97	2.82	Quadro 6000
Proposed sorting algorithm	0.0038	0.016	0.071	0.29	1.2	NVIDIA TESLA K20M

algorithms for 2^{28} elements and larger. Most sorting algorithms [34–38] require 10 GB of memory to attain high performance, while our algorithm only requires 5 GB. One explanation for this efficiency is that our algorithm does not require storage for merging the sorted partitions to produce the final completely sorted set; thus, there is no need for this temporary storage.

8 Conclusions and future works

In this paper, we proposed a novel comparison-free sorting algorithm, and associated software implementations for single- and 8-threaded CPUs and many-threaded GPUs. Our algorithm parallelizes the computations such that each thread is assigned

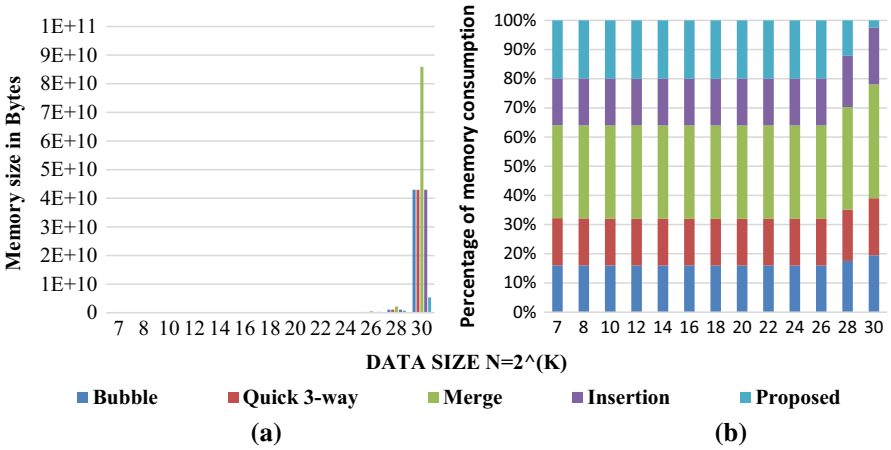


Fig. 18 **a** Total memory consumption in bytes and **b** percentage of memory consumption for our comparison-free sorting algorithm and other common sorting algorithms for varying input set sizes

to individual elements and the workload is well balanced. The software implementations use four forward-flowing computational phases: The first phase assigns the input elements to an index array; the second phase computes the number of occurrences of each element associated with each index and stores this value in the count array; the third phase calculates the accumulated sum at each index of the array and stores this value in the sum array; and the fourth phase correlates the sum and the count arrays to generate the outputted sorted array. Since sorting is performed using only forward-flowing computations without increasing the memory requirements due to storing temporary data and there are no long waiting periods for the data elements for repeated comparisons, these characteristics obviate the need for large global and local memories, which is considered as a bottleneck in most sorting algorithms. The synchronization between the computational phases is maintained using barrier synchronization, and atomic operations are only used when transitioning from phase one to phase two; thus, the computations are fast and operate in parallel.

Our results evaluate different sorting scenarios with different data distributions, number of repeated elements, and input set sizes. Results show increased performance of approximately $4.6\times$ to $6\times$, $4\times$, and $3.5\times$ for our single-threaded CPU, multi-threaded CPU, and GPU implementations, respectively, as compared to most common sorting algorithm, and show reduced runtime variation for different input set sizes, especially for the GPU for large input sizes, which is a characteristic that is not inherent in many other sorting algorithms. Our algorithm also shows a true linear computation rate complexity on the order $O(N)$ with a minimal constant scaling factor for data sizes of the order 2^N and logarithmic timescale with a slope rate less than 1, as compared to other sorting algorithms with linear and logarithmic scale slopes with rate factors ranging between 10 and 100.

Our future work includes leveraging our sorting algorithm for different varieties of commercially available parallel processing computational powers, such as the wide spectrum of GPU-based machines available in the market. This further extends the

performance advantages of our sorting algorithm to big data and reduces any adverse memory effects, further enhancing the processing time for big data.

Acknowledgements The authors would like to acknowledge the support of National Science Foundation (CNS-0953447 and CNS-1718033) and Jordan University of Science and Technology for both providing the financial support to complete this work. The authors would like also to thank the computing center of Synchrotron Light for Experimental Science and Applications in the Middle East (SESAME) for providing the GPU platform resources for our simulations. Sincere thanks are given to Computing Engineers Mustafa A. Alzu'bi and Salman Matalgah for their continuous help in maintaining the GPU resources at the center and providing us with unrestricted access.

References

1. Busse LM, Chehreghani MH, Buhmann JM (2012) The information content in sorting algorithms. In: IEEE International Symposium on Information Theory Proceedings (ISIT), Cambridge, MA, USA, pp 2746–2750
2. Canaan C, Garai MS, Daya M (2011) Popular sorting algorithms. World Appl Program 1(1):62–71
3. Knuth DE (2011) The art of computer programming. Addison-Wesley Professional, Boston
4. Henglein F (2009) What is a sorting function? J Logic Algebr Program 78(7):552–572
5. Ionescu MF, Schauser KE (1997) Optimize parallel bitonic sort. In: Proceedings 11th International Parallel Processing Symposium, Genva, pp 303–309
6. Fuguo D (2010) Several incomplete sort algorithms for getting the median value. Int J Digit Content Technol Appl 4(8):193–198
7. Cederman D, Tsigas P (2009) GPU-quicksort: a practical quicksort algorithm for graphics processors. ACM J Exp Algorithm (JEA) 14(4):1–22
8. Zhang R, Wei X, Watanabe T (2013) A sorting-based IO connection assignment for flip-chip designs. In: 2013 IEEE 10th International Conference on ASIC (ASICON), Shenzhen, pp 1–4
9. Han Y (2004) Deterministic sorting in O(n(log(log n))) time and linear space. J Algorithms Sci Direct Publ 50:602–608
10. Bentley JL, Sedgewick R (1997) Fast algorithms for sorting and searching strings. In: Proceedings of the Eighth annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97), pp 360–369
11. Lin T-C, Kuo C-C, Hsieh Y-H, Wang B-F (2009) Efficient algorithms for the inverse sorting problem with bound constraints under the norm and the Hamming distance. J Comput Syst Sci 75:451–464
12. Abdel-hafeez S, Gordon-Ross A, Abubaker S (2016) A comparison-free sorting algorithm on CPUs. In: 13th International Conference on Applied Computing (AC), Mannheim, Germany, October 2016, pp 3–10
13. Capannini G, Silvestri F, Baraglia R (2012) Sorting on GPUs for large scale datasets: a through comparison. Int Process Manag 48:903–917
14. Abdel-hafeez S, Gordon-Ross A (2017) An efficient O(N) comparison-free sorting algorithm. J IEEE Trans Very Large Scale Integr (VLSI) Syst 2(5):1930–1942
15. Herruzo E, Ruiz G, Benavides J, Plata O, (2007) A new parallel sorting algorithm based on odd-even mergesort. In: 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Naples, (PDP 07), pp 18–22
16. Inoue H, Moriyama T, Komatsu H, Nakatani T (2007) AA-SORT: a new parallel sorting algorithm for multi-core SIMD processors. In: 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, pp 189–198
17. Satish N, Kim C, Chhugani J, Nguyen A, Lee V, Kim D, Dubey P (2010) Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD Sort. In: SIGMOD'10, Indiana, vol 27, number 3, pp 351–362
18. Kundeti V, Rajasekaran S (2011) Efficient out-of-core sorting algorithms for the parallel disks model. J Parallel Distrib Comput 71:1427–1433
19. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore GPUs. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, Rome, pp 1–10
20. Garg A, Goswami S, Garg V (2016) CutShort: a hybrid sorting technique. In: 2016 International Conference on Computing, Communication and Automation (ICCCA), Noida, pp 139–142

21. Yan J-T (1999) An improved optimal algorithm for bubble-sorting based non-Manhattan channel routing. IEEE Trans Comput Aided Des Integr Circuits Syst 18(2):163–171
22. Sareen P (2013) Comparison of sorting algorithms (on the basis of average case). IJARCSSE 3(3):522–532
23. Xiao L, Zhang X, Kubricht SA (2000) Improving memory performance of sorting algorithms. ACM J Exp Algorithm 5(3):1–20
24. Bunse C, Hopfner H, Roychoudhury S, Mansour E (2009) Choosing the BEST Sorting Algorithm from Optimal Energy Consumption. In: ICSOFT 2. INSTICC Press, pp 199–206
25. Mishra AD, Garg D (2008) Selection of best sorting algorithm. Int J Intell Inf Process 2:363–368
26. Kirk DB, Hwu W-MW (2013) Programming massively parallel processors: a hands-on approach, Ch 6. Morgan Kaufman, San Francisco, pp 140–144
27. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing. Pearson Education Limited, Edinburgh
28. Bryant RE, O'Hallaron DR (2003) Computer systems: a programmer's perspective. Pearson Education Inc, Edinburgh
29. Sorting Algorithms Animations. https://www.toptal.com/developers/sorting-algorithms/. Accessed 16 Mar 2016
30. Sareen Pankaj (2013) Comparison of sorting algorithms (on the basis of average case). IJARCSSE 3(3):522–532
31. Thorup Mikkel (2002) Randomized sorting in O(n log log n) time and linear space using addition, shift, and bit-wise boolean operations. Elsevier J Algorithms 42(2):205–230
32. Herruzo E, Ruiz G, Benavides JI, Plata O (2007) A new parallel sorting algorithm based on odd-even mergesort. In: 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 07), pp 18–22
33. Inoue H, Moriyama T, Komatsu H, Nakatani T (2007) AA-SORT: a new parallel sorting algorithm for multi-core SIMD processors. In: 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, pp 189–198
34. Cederman D, Tsigas P (2009) GPU-quicksort: a practical quicksort algorithm for graphics processors. ACM J Exp Algorithm (JEA) 14(4):1–22
35. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium, pp 1–10
36. Cederman D, Tsigas P (2008) On sorting and load balancing on GPUs. Distrib Comput Syst ACM SIGARCH Comput Archit News 36(5):11–18
37. Khan FG, Khan OU, Montrucchio B, Giaccone P (2011) Analysis of fast parallel sorting algorithms for GPU architectures. Front Inf Technol 2011:173–178
38. Farmahini-Farahani A, Duwe HJ, Schulte MJ, Compton K (2013) Modular design of high-throughput, low-latency sorting units. IEEE Trans Comput 62(7):1389–1402