Hindawi
Scientific Programming
Volume 2018, Article ID 2037272, 10 pages
https://doi.org/10.1155/2018/2037272

Research Article

On the Parallelization of Stream Compaction on a Low-Cost SDC Cluster

Gregorio Bernabé and Manuel E. Acacio

Computer Engineering Department, University of Murcia, Murcia, Spain

Correspondence should be addressed to Gregorio Bernabé; gbernabe@um.es

Received 27 February 2018; Accepted 17 July 2018; Published 23 August 2018

Academic Editor: Harald Köstler

Many highly parallel algorithms usually generate large volumes of data containing both valid and invalid elements, and high-performance solutions to the stream compaction problem reveal extremely important in such scenarios. Although parallel stream compaction has been extensively studied in GPU-based platforms, and more recently, in the Intel Xeon Phi platform, no study has considered yet its parallelization using a low-cost computing cluster, even when general-purpose single-board computing devices are gaining popularity among the scientific community due to their high performance per $ and watt. In this work, we consider the case of an extremely low-cost cluster composed by four Odroid C2 single-board computers (SDCs), showing that stream compaction can also benefit—important speedups can be obtained—from this kind of platforms. To do so, we derive two parallel implementations for the stream compaction problem using MPI. Then, we evaluate them considering varying number of processes and/or SDCs, as well as different input sizes. In general, we see that unless the number of elements in the stream is too small, the best results are obtained when eight MPI processes are distributed among the four SDCs that conform the cluster. To add value to the obtained results, we also consider the execution of the two parallel implementations for the stream compaction problem on a very high-performance but power-hungry 18-core Intel Xeon E5-2695 v4 multicore processor, obtaining that the Odroid C2 SDC cluster constitutes a much more efficient alternative when both resulting execution time and required energy are taken into account. Finally, we also implement and evaluate a parallel version of the stream split problem to store also the invalid elements after the valid ones. Our implementation shows good scalability on the Odroid C2 SDC cluster and more compensated computation/communication ratio when compared to the stream compaction problem.

1. Introduction

Continuous improvements in the technologies used to build computers have recently made possible the fabrication of extremely low-cost general-purpose single-board computing devices. Nowadays, one can buy one of these *tiny* computers for a few dollars and make it run Windows 10 or Ubuntu-Linux operating systems [1, 2]. Among the variety of vendors providing these single-board computers (SBCs), maybe the most renowned ones are Raspberry Pi and Odroid. Although the initial aim of these devices was to promote the teaching of basic computer science in schools [3, 4] and developing countries [5–7], recent appearance of single-board computers with multicore ARM CPU chips and several gigabytes of main memory also provides a desirable hardware platform for the project-based learning paradigm in computer science and engineering education [8–11] and have attracted interest of a multitude of projects trying to take advantage of their very low-cost performance ratio (i.e., for scientific computing [12–14]) in contrast with other energy-efficient but which are alternatives of higher cost [15].

Whereas Raspberry Pi SBCs seem to have put the focus more on a "stand-alone" scenario, Odroid devices provide increased processor frequency, more main memory, and higher bandwidth Ethernet capabilities. Particularly, the Raspberry Pi 3 model B that was launched in February 2016 features a 1.2 GHz, 4-core ARM Cortex-A53 CPU chip, 1 GB main memory, and a 10/100 Ethernet port. Compared with its predecessor, the Raspberry Pi 2 model B released in February 2015 adds wireless connectivity (2.4 GHz Wi-Fi

802.11n and Bluetooth 4.1). On the contrary, the Odroid C2 sacrifices wireless connectivity in favor of higher clock frequencies (1.5 GHz, 4-core ARM Cortex-A53 CPU chip), larger main memory (2 GB), and Gigabit Ethernet connection. These characteristics make these particular devices more appropriate at building high-performance low-cost clusters able to meet the demands of some scientific applications.

On the other hand, a common characteristic found in many highly parallel algorithms is that they usually generate large volumes of data containing both valid and invalid elements. In these scenarios, high-performance solutions to the data reduction problem are extremely important. Stream compaction (also known as stream reduction) has been proposed to "compact" an input stream mixed with both valid and invalid elements to a subset with only the valid elements [16]. This way, stream compaction is found in many applications that go from data mining and machine learning (in order to prune invalid nodes after each parallel breadth-first tree traversal step [17]) to deferred shading (to obtain the subset of pixels whose rays intersect, which allows for better workload balancing among the participating threads [18, 19]) or more specifically to speedup dosimetric computations for radiotherapy, using Monte Carlo methods (they compacted computations on photons that worked longer than others [20]) and during voxelization of surfaces and solids [21].

Formally, given a list of elements $i_1, i_2, \ldots, i_n$ belonging to the set **I** and a predicate function stream $F : \mathbf{I} \rightarrow \{true, false\}$, stream compaction divides **I** into valid and invalid elements (ones that satisfy the predicate F and others that do not) and keeps the relative order for all the valid elements in the output (**O**) [18]. As shown in Algorithm 1, the serial stream compaction of **I** under the predicate function F is $O = \{i \varepsilon \mathbf{I} \mid F(i) = true\}$. Therefore, the output **O** simply contains all valid elements copied from the input **I**. An example of the execution of Algorithm 1 can be observed in Figure 1. The list of input elements is composed by numbers between 0 and 4. The serial stream compaction selects all elements that are not zero (assuming that zero represents the invalid value), based on the predicate function F, as shown in the low part of Figure 1. Although Algorithm 1 is simple, the parallelization is not trivial because the output position of each valid element cannot be obtained until all its preceding elements have been discovered [22].

Parallel stream compaction has been extensively studied in GPU-based platforms [16, 18, 22–25], and more recently, parallel implementations for the Intel Xeon Phi processor have also been proposed [26]. In this work, we consider the case of an extremely low-cost cluster composed by four Odroid C2 single-board computers (SDCs), showing that stream compaction can also benefit—important speedups can be obtained—from this kind of platforms. To do so, we derive two parallel implementations for the stream compaction problem using MPI. Then, we evaluate them considering varying number of processes and/or SDCs, as well as different input sizes. In general, we see that unless the number of elements in the stream is too small, the best results are obtained when 8 MPI processes are distributed among the 4 SDCs that conform the cluster.

```
Input: Vector I of length n
Input: Predicate function F
Output: Vector O of valid elements
Output: nvalid: the number of valid elements
(1) nvalid = 0
(2) for i = 0 to n − 1 do
(3)   if F(I[i]) then
(4)     O[nvalid + +] = I[i]
(5)   end if
(6) end for
```

Algorithm 1: Serial stream compaction.

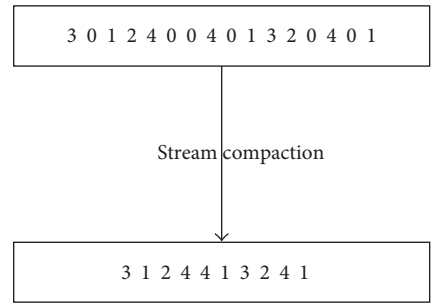

Figure 1: Example of serial stream compaction (zero value is used to represent the invalid elements).

This manuscript extends a preliminary version of this work [27] by making the following two important additional contributions:

(i) To highlight the importance of our study, we also consider the execution of the two parallel implementations for the stream compaction problem on a very high-performance but power-hungry 18-core Intel Xeon CPU E5-2695 v4. Overall, the obtained results show that the Odroid C2 SDC cluster constitutes an appealing alternative to a traditional high-end multicore processor in those contexts in which both low-cost and energy efficiency requirements are present.

(ii) We derive a parallel version of the stream split problem to append the invalid elements to the output stream of the valid elements. We evaluate it on the Odroid C2 SDC cluster, observing good results in terms of scalability that lead to important speedups, and better balance between computation and communication requirements than in the stream compaction problem.

The rest of the paper is organized as follows. The parallelization strategies that we have implemented and evaluated for the stream compaction problem in this work are explained in Section 2. In Section 3, we give the details of the cluster of Odroid C2 SDCs used for the evaluation, and then, we present the experimental results. The parallelization of the stream split problem and results on the Odroid C2 SDCs are exposed in Section 4. Finally, Section 5 draws some important conclusions obtained from this work.

2. Parallel Stream Compaction on a Cluster of Odroid C2 SDCs

In this section, we present the two parallelization strategies that we have considered in this work. In both cases, we have implemented them using MPI [28].

2.1. Parallel Stream Compaction. We have based on the implementation proposed in the Thrust library [29] to develop the parallel stream compaction scheme shown in Algorithm 2. A vector of a particular length, the predicate function, the number of processes, and the pid of each process are the inputs. We have divided Algorithm 2 into four phases, namely: *Validation* phase (lines 4–8), *Scan* phase (lines 9–12), *Communication* phase (lines 13–21), and *Scatter* phase (lines 22–26). During the *Validation* phase, the input vector (**I**) is examined in parallel, and taking into consideration the predicate function, each process annotates the validity of each of its assigned elements in array *temp* (representing 1 a valid element and 0 an invalid one). The parallel *Scan* phase needs an additional array (*scan*) to compute the so-called prefix-sum [30–32], where each element is the addition of all its preceding elements excluding itself. So, each process obtains in parallel the number of valid elements (*nvalid*) in its portion of the stream. Following this, in the *Communication* phase, each process, identified by a pid, sends the number of valid elements that it has found to all the processes with higher pids. All the processes, except the first one, receive the number of valid elements and compute the position (*pos*) of the first of their valid elements. Finally, during the *Scatter* phase, based on the *scan* and *temp* arrays, all valid elements are transferred from the input array to the output one (**I** and **O**, resp.), preserving the order in which these elements appear in the input array.

Figure 2 shows an example of an execution with four MPI processes for a list of input elements composed by numbers ranging between 0 and 4. In this case, the predicate function F selects all elements that are not zero. Now, the input vector of length 16 positions is divided among the four MPI processes (P0, P1, P2, and P3). All the processes carry out the *Validation* and *Scan* phases in parallel. The position (*pos*) computed by each process is shown below the vector *scan*. Finally, the output **O** is built taking into account the *temp* and *scan* vectors, as well as the *pos*, previously computed.

2.2. Parallel Work-Efficient Stream Compaction. In [26], it is presented a work-efficient stream compaction algorithm aimed at improving the computing complexity of the parallel stream compaction that was shown in Algorithm 2. Again, using MPI, we have developed the parallel version of this work-efficient stream compaction and we show it in Algorithm 3. Now, during the *Validation* phase (lines 5–10), each process saves the validity of each element on the array *scan* and stores the number of valid elements on the vector **V**. Therefore, the additional array of integers (*temp*) needed in Algorithm 2 is no longer necessary. In the *Communication* phase (lines 11–26), all processes except the first one send the number of valid elements to the first process (that with pid 0), which executes the inclusive prefix-sum on vector **V** [31], where each element is the addition of all its preceding elements including itself. Then, each position of the array **V** is sent back to the corresponding process. Following this, each process executes the *Scan* phase (lines

```
     Input: Vector I of length n
     Input: Predicate function F
     Input: Number of processes p
     Input: pid of process
     Output: Vector O of valid elements
     Output: nvalid: the number of valid elements
     Output: pos: position to write
 (1) nvalid = 0
 (2) tamp = n/p
 (3) scan[0 : (tamp − 1)] = 0
 (4) for i = 0 to tamp − 1 in parallel do
 (5)    if F(I[i]) then
 (6)       temp[i] = 1
 (7)    end if
 (8) end for
 (9) for i = 0 to tamp − 1 in parallel do
(10)    scan[i] = scan[i − 1] + temp[i − 1]
(11) end for
(12) nvalid = scan[tamp − 1] + temp[tamp − 1]
(13) for i = pid to p − 1 in parallel do
(14)    Send nvalid to process [i + 1]
(15) end for
(16) if pid > 0 then
(17)    for i = 0 to pid in parallel do
(18)       Receive nvalid[i]
(19)       pos = pos + nvalid[i]
(20)    end for
(21) end if
(22) for i = 0 to tamp − 1 in parallel do
(23)    if temp[i] then
(24)       O[pos + scan[i]] = I[i]
(25)    end if
(26) end for
```

Algorithm 2: Parallel stream compaction.

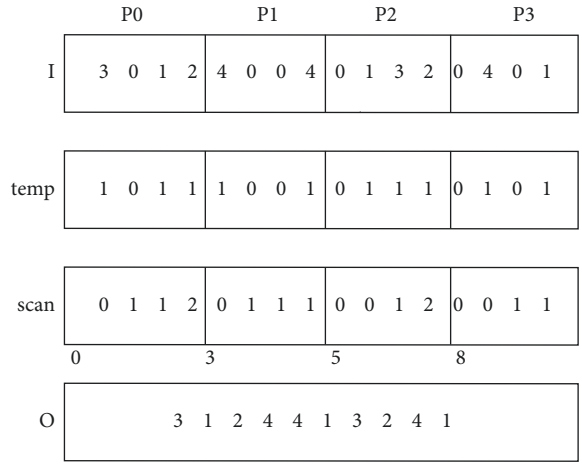

Figure 2: Example of parallel stream compaction.

```
     Input: Vector I of length n
     Input: Predicate function F
     Input: Number of processes p
     Input: pid of process
     Output: Vector O of valid elements
     Output: nvalid: the number of valid elements
 (1) nvalid = 0
 (2) tamp = n/p
 (3) scan[0 : (tamp − 1)] = 0
 (4) V[0 : (t − 1)] = 0
 (5) for i = 0 to tamp − 1 in parallel do
 (6)    if F(I[i]) then
 (7)       scan[i] = 1
 (8)       V[pid] = V[pid] + 1
 (9)    end if
(10) end for
(11) if pid > 0 then
(12)    Send V[pid] to process pid 0
(13) end if
(14) if pid == 0 then
(15)    for i = 1 to npid do
(16)       Receive V[i]
(17)       V[i] = V[i] + V[i − 1]
(18)    end for
(19)    for i = 1 to npid do
(20)       Send V[i − 1] to process pid i
(21)    end for
(22)    nvalid = V[p − 1]
(23) end if
(24) if pid > 0 then
(25)    Receive V[pid − 1]
(26) end if
(27) scan[0] = scan[0] + V[pid − 1]
(28) for i = 0 to tamp − 1 in parallel do
(29)    scan[i] = scan[i − 1] + scan[i]
(30) end for
(31) for i = 0 to tamp − 1 in parallel do
(32)    if scan[i]! = scan[i − 1] then
(33)       O[scan[i − 1]] = I[i]
(34)    end if
(35) end for
```

Algorithm 3: Parallel work-efficient stream compaction.

27–30) on its own segment independently, based on the shifting value received previously. Finally, in the *Scatter* phase (lines 31–35), the validity of each element is rechecked by evaluating two consecutive positions of the *scan* array, obtaining the output array (**O**) with the valid elements from the input array (**I**).

Figure 3 illustrates an example for a list of elements ranging between 0 and 4 and the predicate function F that selects all elements that are not zero for an execution of four MPI processes. As in the previous example, 4 input elements are assigned to each MPI process and the *Validation* phase is applied producing directly the validity of each element on vector *scan* together with the number of valid elements that each process finds out. The latter is stored on vector **V**. Then, the process $P0$ executes the inclusive prefix-sum on vector **V** and sends back the output to the rest of the processes as is indicated by the arrows in Figure 3. Finally, each process

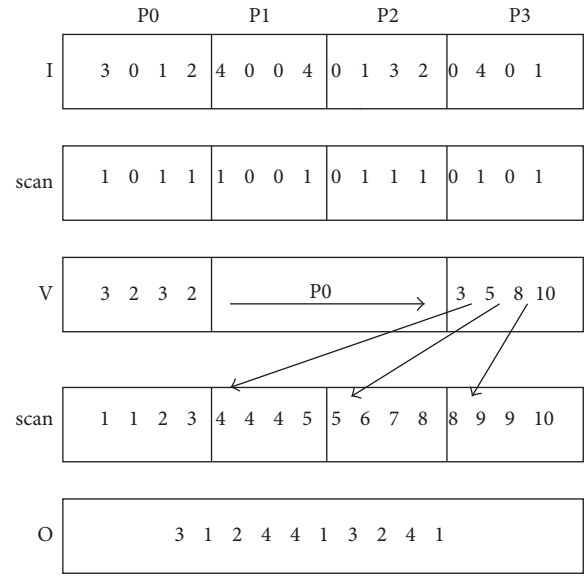

Figure 3: Example of parallel work-efficient stream compaction.

enters the *Scan* and *Scatter* phases taking into account the corresponding shifting value calculated by $P0$.

3. Experiments

We have built a cluster which is composed by four Odroid C2 nodes. Each node contains a 1.5 GHz quad-core 64-bit ARM Cortex-A53 CPU and 2 GB of RAM memory. All the nodes are interconnected through a Gigabit Ethernet switch. The operating system installed on each node is Ubuntu 16.04.02 LTS. In this cluster, we have installed MPICH (v3.2) as the MPI library implementation.

We have executed and measured the two parallelization strategies for stream compaction presented in Section 2 on this cluster. The baseline for all the comparisons is the sequential version of Algorithm 2 without the *Communication* phase. Moreover, we have configured different parallel execution scenarios for the two parallel versions of the stream compaction problem explained before. We consider parallel executions with 2, 4, 8, and 16 MPI processes, running on the same Odroid C2 board or different boards (up to 4). We have chosen several input data sizes for our tests. In particular, we consider input arrays with 1M, 8M, 32M, and 64M integer elements ranging between 0 and 4. The predicate function in all cases determines as valid all numbers that are not zero. The 64M input set is the largest configuration that we could run taking into account the 2 GiB limit that the Odroid SDC imposes.

3.1. Execution Time Results. Figures 4(a), 4(b), 5(a), and 5(b) show the execution times (in milliseconds) that are observed for input data sizes of 1M, 8M, 32M, and 64M elements, respectively. For all these figures, from left to right, we first present the result obtained for the sequential version (Sequential), and then we show the results for the parallel stream compaction (Compaction) and parallel work-efficient stream compaction (Compaction-Shifted) parallelization strategies, respectively. For each one of them, we consider 2, 4, and 8

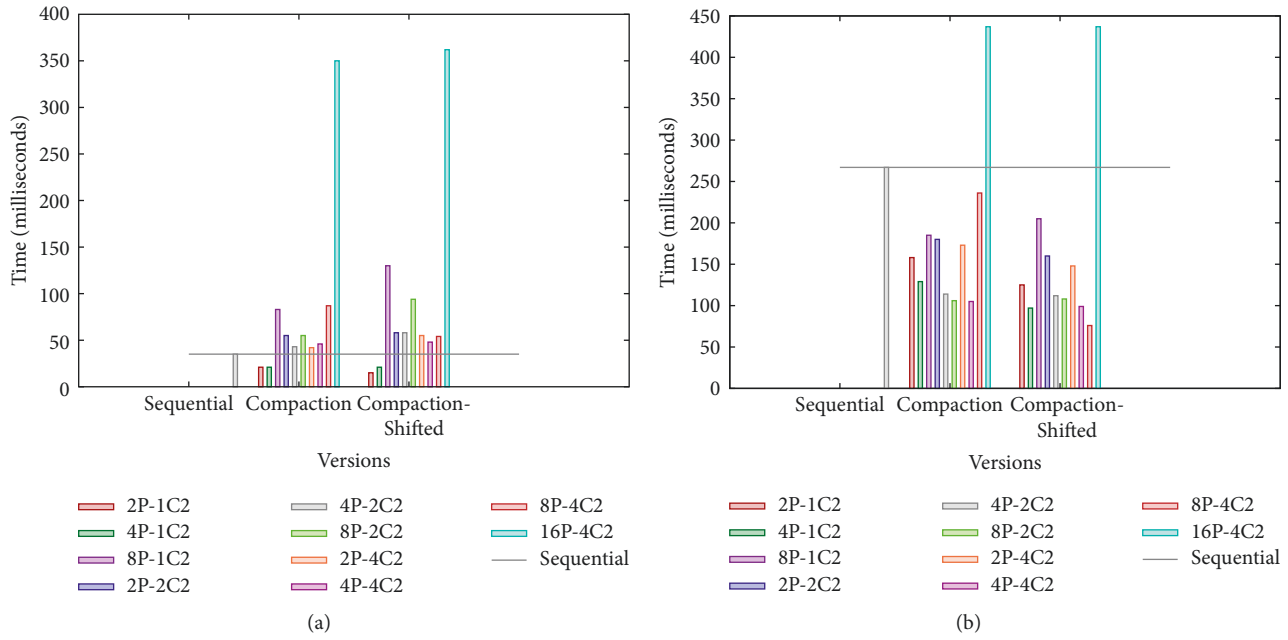

Figure 4: Execution times (milliseconds) for stream compaction: (a) 1M elements and (b) 8M elements.

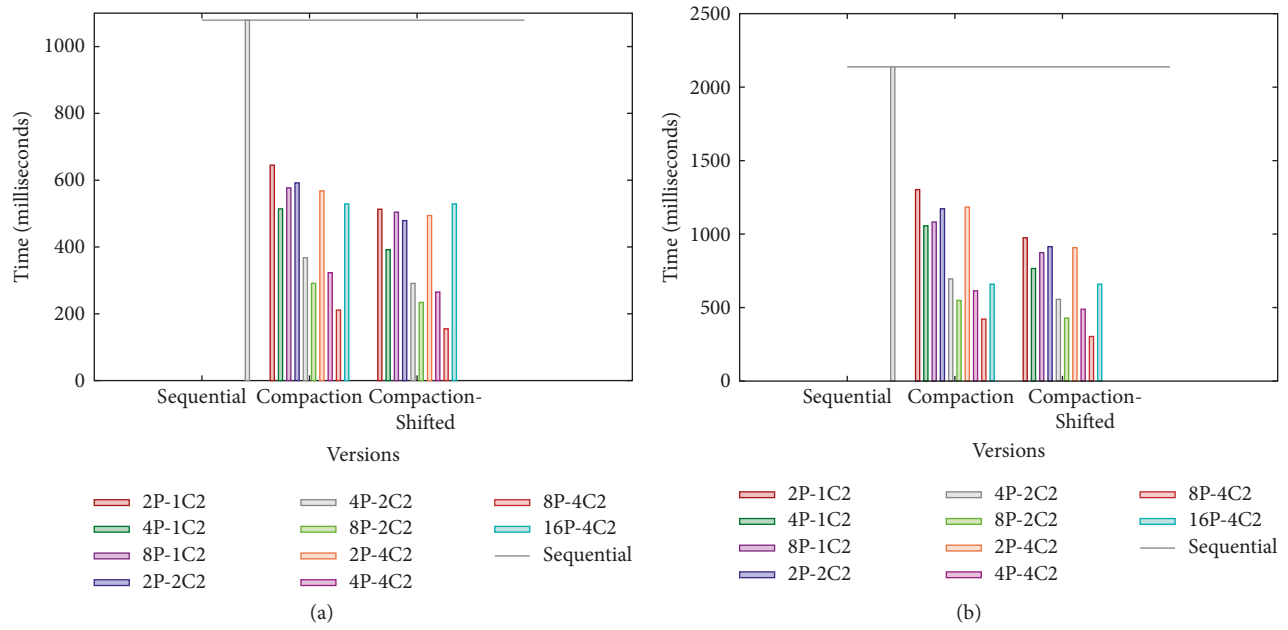

Figure 5: Execution times (milliseconds) for stream compaction: (a) 32M elements and (b) 64M elements.

MPI processes running on one Odroid C2 board (2P-1C2, 4P-1C2, and 8P-1C2, resp.) and on 2 Odroid C2 boards, having 1, 2, and 4 MPI processes per board in each case (2P-2C2, 4P-2C2, and 8P-2C2, resp.), and finally 2, 4, 8, and 16 MPI processes running on 4 Odroid C2 boards, having 1, 2, or 4 processes per board as appropriate (2P-4C2, 4P-4C2, 8P-4C2, and 16P-4C2, resp.).

From Figure 4(a), we can see that the two proposed parallelization strategies for the stream compaction problem obtain noticeable speedups when they are executed on a single Odroid C2 board with 2 or 4 MPI processes with regard to the sequential version. However, the executions on different Odroid C2 boards show negative outcomes from the performance point of view when the size of the input array is excessively small (1M elements). What makes the differences is that in the first case, all communications take place on the same board and, therefore, can be performed with low latency. Contrarily, what happens when

communications involve several Odroid C2 boards? In this case, the time required for communication does not compensate the small processing time that is needed to obtain the stream compaction for such a small number of elements (the communication time constitutes between 65% and 92% of the execution time). Moreover, the executions on a single Odroid C2 SDC with 8 MPI processes (2 MPI processes per core) also show negative speedups, revealing (as expected) that a configuration with more than one MPI process per core increments the communications, which represents up to 58% of the total time, and potentially slows computations.

Taking a closer look at the results for one Odroid C2 board and 1M input size, we see that the speedups of the parallel stream compaction strategy for 2 and 4 MPI processes are 1.68 and 1.64, respectively. Similarly, the work-efficient stream compaction parallelization strategy obtains speedups of 2.25 and 1.63 for 2 and 4 MPI processes, respectively. Therefore, in both cases, fewer MPI processes, and therefore, less amount of communications among several processes, bring the best results. These two parallel versions do not scale due to the small computation/communication ratio that they exhibit (approximately 4 and 2 for 2 and 4 processes for both proposals), which decreases as the number of processes grows.

In general, from Figures 4(b), 5(a), and 5(b), we can see that as the input data size increases, so it does the speedups obtained by the two parallelization strategies analyzed in this work when more cores are involved. The exception is the configuration with 16 processes running on 4 Odroid C2 boards (4 processes per board), which reaches lower speedups than that with 8 processes running on 4 Odroid C2 boards (2 processes per board).

More specifically, Figure 4(b) shows the results seen for 8M elements. In this case, the two proposed parallelization strategies obtain significant speedups when executed on a single Odroid C2 board with 2, 4, or 8 MPI processes with regard to the sequential version. Additionally, the scalability is good for 2 and 4 MPI processes obtaining 1.69 and 2.06 for the parallel stream compaction strategy and 2.14 and 2.74 for the parallel work-efficient stream compaction approach. Therefore, for medium input data sizes, the computation/communication ratio is appropriate (approximately 100 and 7 for 2 and 4 processes). Although the two parallelization strategies also achieve gains for the configuration (8P-1C2) with 2 processes per core on a single Odroid C2 (speedups of 1.44 and 1.31, resp.), these speedups are (as expected) lower than those of the (4P-1C2) case. It is clear that the fact that there are twice the number of MPI processes than the total number of cores available introduces extra scheduling overhead and causes worse use of cores' resources (such as caches). On the other hand, the executions on different Odroid C2 SDCs (except for 8P-4C2 and 16P-4C2) present important speedups and good scalability for 2, 4, and 8 MPI processes for the two proposed parallelization strategies. Thus, the increment in the number of processes per Odroid C2 implies a suitable operation of the Odroid C2 cluster, where the communication latency among the different boards of the cluster does not ballast performance. In the 8P-4C2 case is where the performance differences between the two parallelization strategies start appearing. Whereas the most efficient strategy (namely, Compaction-Shifted) achieves the highest speedup for this configuration, the other one cannot improve over the results reached by 4P-4C2 demonstrating its more limited scalability for medium-sized workloads. Finally, the large number of processes involved in 16P-4C2 results in excessively small computation/communication ratios, which is the reason for the negative outcomes observed in both cases (the fraction of time due to communications reaches 87%).

As we can observe in Figures 5(a) and 5(b), having higher input data sizes for the two parallel stream compaction strategies results in significant gains in all the configurations. For both input data sizes, both Compaction and Compaction-Shifted obtain speedups that are close to that observed for the 8M element case when executed on a single Odroid C2 SDC with 2, 4, or 8 MPI processes. However, the resulting speedups become even more important as the number of involved cores grows. Moreover, they scale nicely for 2, 4, and 8 MPI processes, achieving their highest values for 8 MPI processes running on 4 Odroid C2 SDCs (5.10 and 5.06 for the parallel stream compaction and input data sizes of 32M and 64M, resp., and 6.96 and 7.04 for the parallel work-efficient stream compaction and input data sizes of 32M and 64M, resp.). It is also worth noting that even for these large input sizes, the results reached for the 16P-4C2 configuration are worse than those of the 8P-4C2 in both cases. Now the differences between them become narrower as input data sizes increase.

3.2. Energy Efficiency Results. To give readers a more complete view that can help put our results in context, we also consider the case of executing the parallel stream compaction and parallel work-efficient stream compaction approaches described in Sections 2.1 and 2.2, respectively, on a conventional, high-performance multicore processor. In particular, we have considered the case of a state-of-the-art Intel® Xeon® E5-2695 v4 multicore processor running at 2.10 GHz. Particularly, the Intel Xeon multicore processor has 18 cores, and its price is approximately 8× that of the complete cluster. We have a dual-socket configuration.

The comparison between the 4 Odroid C2 cluster and the Intel Xeon is done by taking into consideration the execution times of each version on every platform and the reported thermal design power (TDP) measures in each case (16 W for the Odroid C2 and 120 W for Intel Xeon processor). We have measured the energy consumption using RAPL [33] in the Intel Xeon processor. Although for the 1M input data size resulting watts are lesser than 120, this TDP is overcome in the rest of input data sizes. Therefore, we have used the TDP as an average measure of the energy consumption.

Figures 6 and 7 show total energy consumption (in joules) for parallel stream compaction and parallel work-efficient stream compaction, respectively. Again, results for input data sizes of 1M, 8M, 32M, and 64M elements are reported. In both figures, we show the results for 2, 4, 8, and 16 MPI processes running on 4 Odroid C2 boards (having 1, 2, or 4 processes per board as appropriate (2P-OC2, 4P-OC2,

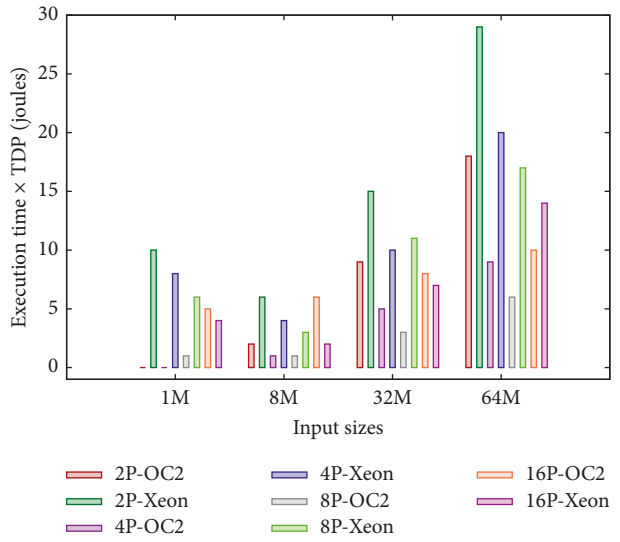

Figure 6: Execution time × TDP for parallel stream compaction for the 4 Odroid C2 cluster and Intel Xeon processor.

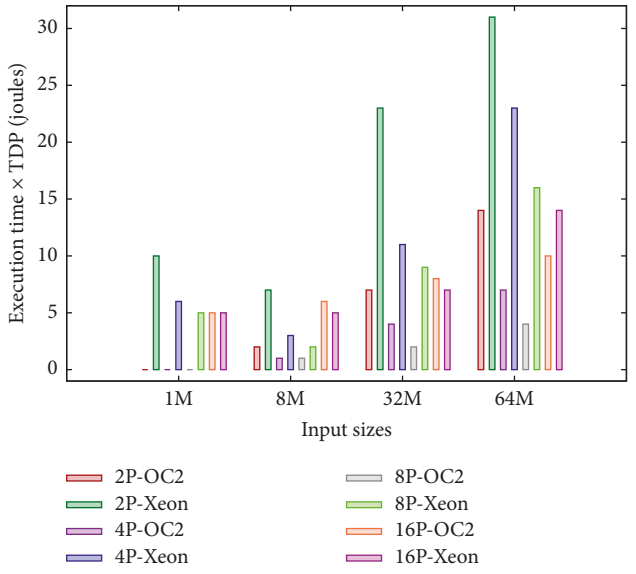

Figure 7: Execution time × TDP for parallel work-efficient stream compaction in 4 Odroid C2 boards and Intel Xeon.

Input: Vector **I** of length n
Input: Predicate function F
Input: Number of processes p
Input: pid of process
Output: Vector **O** of valid elements
Output: $nvalid$: the number of valid elements

(1) $nvalid = 0$
(2) $tamp = n/p$
(3) $scan[0 : (tamp - 1)] = 0$
(4) $V[0 : (t - 1)] = 0$
(5) **for** $i = 0$ to $tamp - 1$ in parallel **do**
(6) **if** $F(\mathbf{I}[i])$ **then**
(7) $scan[i] = 1$
(8) $V[pid] = V[pid] + 1$
(9) **end if**
(10) **end for**
(11) **if** $pid > 0$ **then**
(12) Send $V[pid]$ to process pid 0
(13) **end if**
(14) **if** $pid == 0$ **then**
(15) **for** $i = 1$ to $npid$ **do**
(16) Receive $V[i]$
(17) $V[i] = V[i] + V[i - 1]$
(18) **end for**
(19) **for** $i = 1$ to $npid$ **do**
(20) Send $V[i - 1]$ to process pid i
(21) **end for**
(22) $nvalid = V[p - 1]$
(23) Send $nvalid$ to all processes
(24) **end if**
(25) **if** $pid > 0$ **then**
(26) Receive $V[pid - 1]$
(27) Receive $nvalid$
(28) **end if**
(29) $scan[0] = scan[0] + V[pid - 1]$
(30) **for** $i = 0$ to $tamp - 1$ in parallel **do**
(31) $scan[i] = scan[i - 1] + scan[i]$
(32) **end for**
(33) **for** $i = 0$ to $tamp - 1$ in parallel **do**
(34) **if** $scan[i]! = scan[i - 1]$ **then**
(35) $\mathbf{O}[scan[i - 1]] = \mathbf{I}[i]$
(36) **else**
(37) $\mathbf{O}[nvalid + (i - scan[i - 1])] = \mathbf{I}[i]$
(38) **end if**
(39) **end for**

Algorithm 4: Parallel stream split.

8P-OC2, and 16P-OC2, resp.)) and running on the Intel Xeon using 2, 4, 8, and 16 cores (2P-Xeon, 4P-Xeon, 8P-Xeon, and 16P-Xeon, resp.).

In the figures, we can see that the trend observed for the low-cost SDC cluster does not keep in the case of the Intel Xeon, and the best results in this case are obtained for 16 cores. The fact that, in this case, all communications occur on the same chip significantly reduces the overhead of involving a larger number of cores in the computation. This is also evidenced by the fact that speedups are obtained even for the small problem sizes. However, even when the computational power of the Intel Xeon is much greater than the one of the Odroid C2 clusters, the very large TDP of the Intel Xeon ballasts its results when energy efficiency is also considered. Particularly, the best results for the Odroid C2 cluster (obtained when 2 processes run on 4 boards) clearly outperform those achieved when 16 processes are executed using 2 Intel Xeon chips, demonstrating that the Odroid C2 SDC cluster constitutes an appealing alternative to a traditional high-end multicore processor in those contexts in which both low-cost and energy efficiency requirements are found.

4. Extension to Stream Split

There are some applications, for example, a radix sort [34] or random forest-based data classifiers [35], in which it is

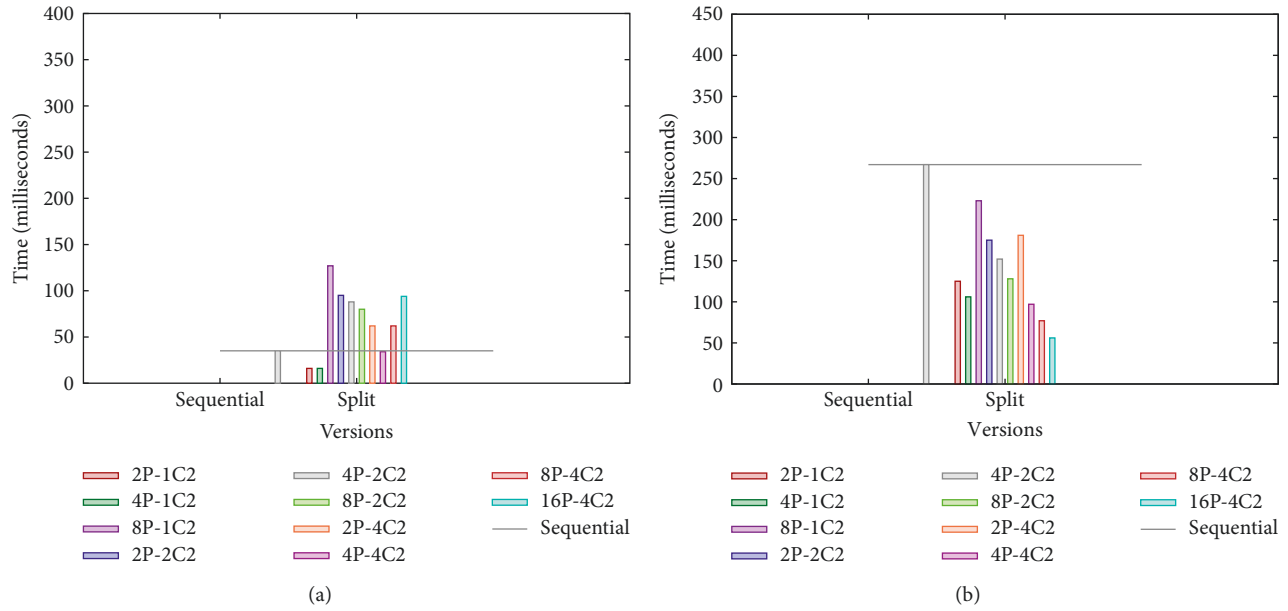

Figure 8: Execution times (milliseconds) for stream split: (a) 1M elements and (b) 8M elements.

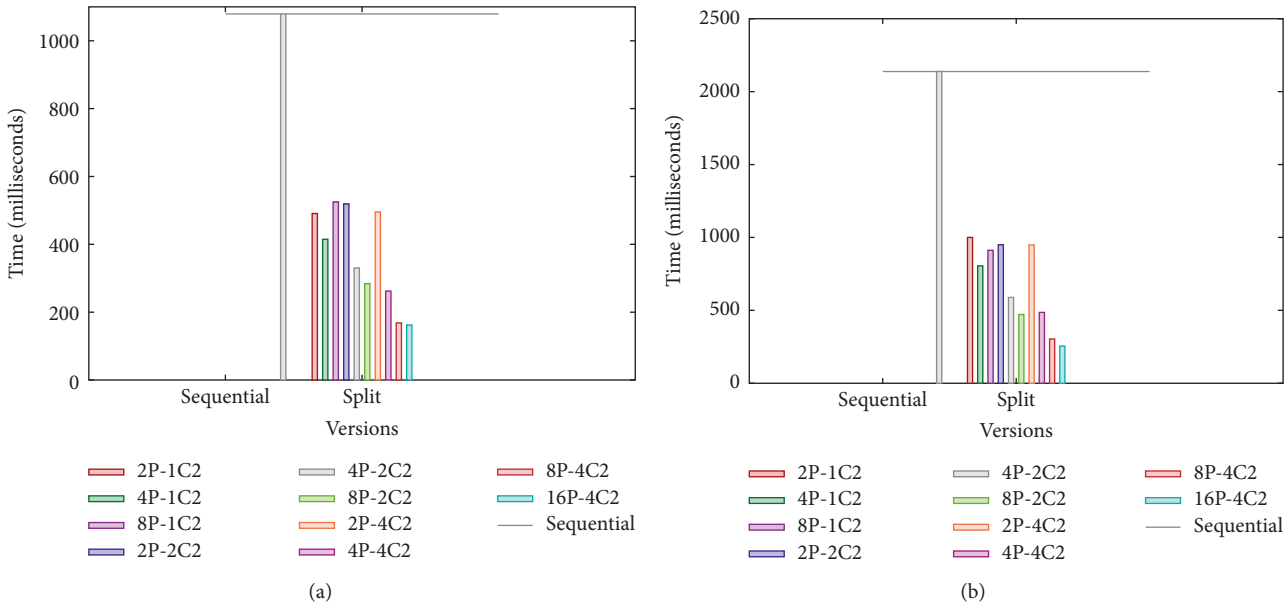

Figure 9: Execution times (milliseconds) for stream split: (a) 32M elements and (b) 64M elements.

needed to append the invalid elements to the end of the output stream with the valid elements. This is the so-called stream split problem. In this work, we have also developed a parallel solution to the stream split problem and we present it in Algorithm 4. The stream split algorithm is very much like the parallel work-efficient stream compaction version presented in Algorithm 3. The main differences are that now the first process must send the number of valid elements to all the processes with higher pids (line 23), the different processes (except the first one) receive the number of valid elements (line 27), and if an element is invalid, it would be stored after the valid elements plus an offset given by $i - scan[i-1]$, as we can observe in lines 36–37.

Figures 8(a), 8(b), 9(a), and 9(b) show the execution times (in milliseconds) that are observed for input data sizes of 1M, 8M, 32M, and 64M elements, respectively. For all these figures, from left to right, we first present the results obtained for the sequential version (Sequential), and then we show the results for the parallel stream split (Split). For each one of them, we consider 2, 4, and 8 MPI processes running on one Odroid C2 board (2P-1C2, 4P-1C2, and 8P-1C2, resp.) and on 2 Odroid C2 boards, having 1, 2, and 4 MPI

processes per board in each case (2P-2C2, 4P-2C2, and 8P-2C2, resp.), and finally 2, 4, 8, and 16 MPI processes running on 4 Odroid C2 boards, having 1, 2, or 4 processes per board as appropriate (2P-4C2, 4P-4C2, 8P-4C2, and 16P-4C2, resp.).

In general, from Figures 8(a), 8(b), 9(a), and 9(b), we can see that the trend observed for the different input sizes is very similar to that already explained for the Compaction and Compaction-Shifted proposals except for the configuration with 16 processes running on 4 Odroid C2 boards (4 processes per board) and input sizes from 8M to 64M elements, which reaches higher speedups than the rest of configurations and that those observed in the two previous approaches. Speedups of 4.74, 6.66, and 8.36 with regard to the sequential version are achieved for 8M elements, 32M elements, and 64M elements, respectively. Now, the increase in computation due to the storage of the invalid elements compensates the communication requirements and significant speedups are obtained.

More specifically, the scalability is good for 2, 4, and 8 MPI processes running on 2 Odroid C2 boards, obtaining 1.52, 1.75, and 2.08 for the parallel stream split approach for 8M elements. Therefore, for medium input data sizes, the computation/communication ratio is appropriate. On the same way, the scalability is suitable for 2, 4, and 8 MPI processes for higher input data sizes. For example, for 64M elements, the speedups achieved are 2.25, 3.63, and 4.52 for 2, 4, and 8 MPI processes executing on 2 Odroid C2 boards. Moreover, the scalability is good for 2, 4, 8, and 16 MPI processes running on 4 Odroid C2 boards, obtaining 1.47, 2.76, 3.47, and 4.74 for medium input sizes, whereas gains for big input sizes are very similar, and achieving, for instance, 2.25, 4.39, 7.02, and 8.36 for 64M elements.

5. Conclusions

In this work, we have studied the parallelization of the stream compaction problem on a low-cost cluster of single-board computers. Particularly, we have configured the low-cost cluster from 4 Odroid C2 SDCs which are interconnected using a typical Gigabit Ethernet switch. We have implemented two parallel versions for the stream compaction problem using MPI. Then, we evaluate them considering varying number of processes and/or SDCs, as well as different input sizes. In general, we see that when the number of elements in the stream is too small, the most important benefits are observed when all participating processes are in the same Odroid board. In this case, the low computation/communication ratio for small number of input elements cannot make up for the overhead entailed by the inter-SDC communications. As the number of elements in the input stream increases, so it does the number of processes that can participate in parallel executions, and important speedups are reached. Overall, the best results are reached when eight MPI processes are distributed among the four SDCs that conform the cluster. In this case, speedups of 5.10 and 7.04 are obtained for the Compaction and Compaction-Shifted strategies, respectively, for the larger problem size considered in this work (input data size of 64M). Moreover, to add value to the obtained results, we also consider the execution of the two parallel implementations for the stream compaction problem on a very high-performance but power-hungry 18-core Intel Xeon E5-2695 v4 multicore processor, obtaining that the Odroid C2 SDC cluster constitutes a much more efficient alternative when both resulting execution time and required energy are taken into account. Finally, the parallelization of the stream split problem is implemented and evaluated on the Odroid C2 SDC cluster. In this case, for input data sizes starting from 8M elements, important speedups are achieved and the computation/communication is more equilibrated due to the storage of the invalid elements. In summary, the best results are obtained for the configuration of 16 MPI processes running on 4 Odroid C2 boards. In this case, speedups of 6.66 and 8.36 are reached for input data sizes of 32 and 64M elements, respectively.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the Spanish MINECO and by European Commission FEDER funds, under Grant TIN2015-66972-C5-3-R.

References

[1] P. Membrey and D. Hows, *Learn Raspberry Pi 2 with Linux and Windows 10*, Apress, New York City, NY, USA, 2nd edition, 2015.

[2] M. Richardson and S. Wallace, *Getting Started with Raspberry Pi*, O'Reilly Media, Inc., Sebastopol, CA, USA, 2012.

[3] A. Hague, G. Hastings, M. Klling et al., *The Raspberry Pi Education Manual Version 1.0*, Creative Commons License, Computing at School, London, UK, 2012.

[4] M. Kolling, "Educational programming on the raspberry Pi," *Electronics*, vol. 5, no. 3, p. 33, 2016.

[5] R. Heeks and A. Robinson, "Ultra-low-cost computing and developing countries," *Communications of the ACM*, vol. 56, no. 8, pp. 22–24, 2013.

[6] M. Ali, J. H. A. Vlaskamp, N. N. Eddin, B. Falconer, and C. Oram, "Technical development and socioeconomic implications of the Raspberry Pi as a learning tool in developing countries," in *Proceedings of the 2013 5th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 103–108, Colchester, UK, September 2013.

[7] M. Srinivasan, B. Anand, A. A. Venus et al., "Greeneducomp: low cost green computing system for education in rural India: a scheme for sustainable development through education," in *Proceedings of the Global Humanitarian Technology Conference (GHTC 2013), IEEE*, pp. 102–107, San Jose, CA, USA, October 2013.

[8] X. Zhong and Y. Liang, "Raspberry Pi: an effective vehicle in teaching the internet of things in computer science and engineering," *Electronics*, vol. 5, no. 3, p. 56, 2016.

[9] R. F. Bruce, J. D. Brock, and S. L. Reiser, "Make space for the Pi," in *Proceedings of the SoutheastCon, 2015*, pp. 1–6, Fort Lauderdale, FL, USA, April 2015.

[10] K. McCullen, "Teaching embedded systems using the raspberry pi and sense hat," *Journal of College Science Teaching*, vol. 32, no. 6, pp. 200–202, 2017.

[11] C. Williams and S. Kurkovsky, "Raspberry Pi creativity: a student-driven approach to teaching software design patterns," in *Proceedings of the 2017 IEEE Frontiers in Education Conference (FIE)*, pp. 1–9, Indianapolis, IN, USA, October 2017.

[12] D. Abdurachmanov, P. Elmer, G. Eulisse, and S. Muzaffar, "Initial explorations of ARM processors for scientific computing," *Journal of Physics: Conference Series*, vol. 523, no. 1, article 12009, 2014.

[13] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J. F. Mhaut, "Performance/energy trade-off in scientific computing: the case of arm big.little and intel sandy bridge," *IET Computers and Digital Techniques*, vol. 9, no. 1, pp. 27–35, 2015.

[14] A. Pajankar, *Raspberry Pi Supercomputing and Scientific Programming*, Apress, New York City, NY, USA, 2017.

[15] D. Cesini, E. Corni, A. Falabella et al., "Power-efficient computing: experiences from the COSA project," *Scientific Programming*, vol. 2017, Article ID 7206595, 14 pages, 2017.

[16] D. Roger, U. Assarsson, and N. Holzschuch, "Efficient stream reduction on the GPU," in *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, USA, 2007.

[17] B. Ren, T. Poutanen, T. Mytkowicz, W. Schulte, G. Agrawal, and J. R. Larus, "SIMD parallelization of applications that traverse irregular data structures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–10, Shenzhen, China, February 2013.

[18] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart, "Stream compaction for deferred shading," in *Proceedings of the Conference on High Performance Graphics, HPG'09, ACM*, pp. 173–180, New York, NY, USA, 2009.

[19] K. Garanzha, S. Premoze, A. Bely, and V. A. Galaktionov, "Grid-based SAH BVH construction on a GPU," *Visual Computer*, vol. 27, no. 6–8, pp. 697–706, 2011.

[20] S. Hissoiny, B. Ozell, H. Bouchard, and P. Desprs, "GPUMCD: a new GPU-oriented Monte Carlo dose calculation platform," *Medical Physics*, vol. 38, no. 2, pp. 754–764, 2011.

[21] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on GPUs," *ACM Transactions on Graphics*, vol. 29, no. 6, p. 1, 2010.

[22] A. Pirjan, "Solutions for optimizing the stream compaction algorithmic function using the compute unified device architecture," *Journal of Information Systems and Operations Management*, vol. 5, no. 2, pp. 456–477, 2011.

[23] D. Horn, "Stream reduction operation for GPGPU applications," *GPU Gems*, vol. 2, pp. 573–589, 2005.

[24] S. Sengupta, A. E. Lefohn, and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, Chapel Hill, NC, USA, 2006.

[25] D. M. Hughes, I. S. Lim, M. W. Jones, A. Knoll, and B. Spencer, "InKCompact, InKernel stream compaction, and its application to MultiKernel data visualization on GeneralPurpose GPUs," *Computer Graphics Forum*, vol. 32, no. 6, pp. 178–188, 2013.

[26] Q. Sun, C. Yang, C. Wu, and L. F. Liu, "Fast parallel stream compaction for IA-based multi/many-core processors," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Cartagena, Colombia, May 2016.

[27] G. Bernabé and M. E. Acacio, "Efficient parallel stream compaction on a extremely low-cost SDC cluster," in *Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*, pp. 304–315, Trieste, Italy, July 2017.

[28] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, MIT Press, Cambridge, MA, USA, 2nd edition, 1999.

[29] Nvidia, "Thrust," 2015, http://docs.nvidia.com/cuda/thrust.

[30] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[31] G. E. Blelloch, "Prefix sums and their applications," Tech. Rep. CMU-CS-90–190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[32] W. D. Hillis and G. L. Steele Jr., "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[33] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[34] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the Conference on High Performance Graphics, HPG'09*, pp. 159–166, ACM, New York, NY, USA, 2009.

[35] B. Ren, T. Poutanen, T. Mytkowicz, W. Schulte, G. Agrawal, and J. R. Larus, "Simd parallelization of applications that traverse irregular data structures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), CGO'13*, pp. 1–10, IEEE Computer Society, Washington, DC, USA, 2013.