CORE
Metadata, citation and similar papers at core.ac.uk
Provided by MUCC (Crossref)

Hindawi
Scientific Programming
Volume 2017, Article ID 2040865, 8 pages
https://doi.org/10.1155/2017/2040865

Hindawi

Research Article

Accelerating the HyperLogLog Cardinality Estimation Algorithm

Cem Bozkus[1] and Basilio B. Fraguela[2]

[1] *Bilkent Üniversitesi, Ankara, Turkey*
[2] *Universidade da Coruña, A Coruña, Spain*

Correspondence should be addressed to Basilio B. Fraguela; basilio.fraguela@udc.es

Received 29 June 2017; Accepted 6 August 2017; Published 14 September 2017

Academic Editor: Piotr Luszczek

Copyright © 2017 Cem Bozkus and Basilio B. Fraguela. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In recent years, vast amounts of data of different kinds, from pictures and videos from our cameras to software logs from sensor networks and Internet routers operating day and night, are being generated. This has led to new big data problems, which require new algorithms to handle these large volumes of data and as a result are very computationally demanding because of the volumes to process. In this paper, we parallelize one of these new algorithms, namely, the HyperLogLog algorithm, which estimates the number of different items in a large data set with minimal memory usage, as it lowers the typical memory usage of this type of calculation from $O(n)$ to $O(1)$. We have implemented parallelizations based on OpenMP and OpenCL and evaluated them in a standard multicore system, an Intel Xeon Phi, and two GPUs from different vendors. The results obtained in our experiments, in which we reach a speedup of 88.6 with respect to an optimized sequential implementation, are very positive, particularly taking into account the need to run this kind of algorithm on large amounts of data.

1. Introduction

Very often the processing of very large data sets does not require accurate solutions, being enough to find approximate ones that can be achieved much more efficiently. This strategy, called approximate computing, has been used in computing for many years and can be applied in those contexts where answers that are close enough to the actual value are acceptable, giving place to a trade-off of accuracy for other resources, typically memory space and time. For example, the scholastic gradient descent algorithm of machine learning is used to calculate approximate local minimum and not exact global minimum. Another example is bloom filters [1], which allow easily checking whether an item is in a set or not by using multiple hash functions, there being a certain probability of false positives, that is, of classifying as members of the set items that actually do not belong to it.

HyperLogLog (HLL) [2] is a very powerful approximate algorithm in the sense that it can practically and efficiently give a good estimation of the cardinality of a data set, meaning the number of different items in it with respect to some characteristics. This value has many real life applications, making its computation a must for high profile companies working in big data. This algorithm is used by basically anyone who has a data set and needs its cardinality without wasting space, as it can calculate the cardinality of N items with $O(1)$ space complexity. Because of the way it is implemented, HLL allows many smaller budget companies and individuals without large memory banks to use large data sets.

In this paper we develop two parallel implementations of the HyperLogLog algorithm, one of them based on OpenMP and targeted to multicore processors and Intel Xeon Phi accelerators and another one based on OpenCL, which can be run not only on these systems but also on other kinds of accelerators such as GPUs. Both implementations are compared on an Intel Xeon Phi and a standard multicore system, while the performance of the OpenCL version is evaluated in these platforms as well as in an NVIDIA Tesla K20m GPU and an AMD FirePro S9150 GPU.

The remainder of this paper is organized as follows. Section 2 briefly summarizes the HyperLogLog algorithm and its sequential implementation, while Section 3 details our parallel implementations. This is followed by an evaluation in Section 4. Then, Section 5 is devoted to the related work. Finally, Section 6 is devoted to our concluding ideas.

```
input: Input file inputFile
input: Number of estimator buckets N
data: Vector of N buckets M⃗ = (M_0, ..., M_{N−1})
output: Estimation E of number of different items in the input file
M_index = 0,  0 ≤ index < N
while not at end of file inputFile do
    dataChunk = readChunk(inputFile)
    foreach item in dataChunk do
        hash = hashFunction(item)
        [remainder, index] = splitHash(hash)
        nleadingZeros = countLeadingZeros(remainder)
        M_index = max (M_index, nleadingZeros)
    end
end
E = Alpha × N^2 × (sum_{i=0}^{N−1} 2^{−M_i})^{−1}
```

Algorithm 1: Pseudocode for HyperLogLog.

2. HyperLogLog Algorithm

The HyperLogLog algorithm, thoroughly described in [2], hashes each item in the set to be analyzed, obtaining an associated 32-bit binary number. This value is then decomposed in two parts. The first b bits of this value are used to determine which bucket of the estimator this number falls in out of the $N = 2^b$ ones available, and in the rest of the bits we count the amount of leading zeros to estimate the probability of this number occurring. Every estimator bucket M_j, $0 \leq j < 2^b$, stores the maximum amount of leading zeros found for all the values associated with that bucket. After all the items are processed in this way, the harmonic mean of the value 2^{M_j} for all the buckets is calculated. This value is then multiplied by a constant Alpha and the amount of buckets N. This is the raw HyperLogLog estimate, which can be expressed as

$$E = \text{Alpha} \times N^2 \times \left(\sum_{j=1}^{N} 2^{-M_j} \right)^{-1} \quad (1)$$

and is usually the correct answer, although in some situations corrections are applied. Namely, if $E < 2.5N$ and there are estimated buckets with the value 0, then the final estimation is $N \times \log(N/V)$, where V is the number of buckets equal to 0. Similarly, if $E > (2^{32}/30)$ the final estimation must be $-2^{32}\log(1 - E/2^{32})$, although this latter correction does not apply if binaries of 64 bits instead of 32 are used for the result of the hash function. Either way, we can see that the computationally intensive part of the algorithm is the derivation of the hashes for the items and the update of the buckets.

A sequential implementation for this algorithm is shown in Algorithm 1. Since the algorithm is usually applied to large data sets, the data to process is found in an input stream that is processed by chunks, which are groups of consecutive data taken from the stream, in the main loop of the algorithm. The innermost loop takes care of the processing of each item in the current chunk. Each item is first hashed. The resulting value is split into two pieces. The first part is a binary number that represents the index of the estimator array. For example, when using 256 buckets, the index field corresponds to the 8 less significant bits of the hash, and the value 00010011 is the bucket number 35. The rest of the bit stream is given to a function that counts the leading zeros and then the associated bucket is updated with this value if it is larger than the current value in the bucket. When the process finishes, (1) is applied to compute the raw estimate. Figure 1 presents the algorithm flow chart.

3. Parallel Implementations

As a first step we developed a sequential implementation of HyperLogLog used as input items 32 bit integers, so that its purpose was the estimation of the number of different integers in an input stream. Before proceeding to the development of parallel implementations we optimized our sequential implementation. This step was critical, as the initial implementation of the hash function was based on a temporary C++ string (`std::string`), which implied allocations and deallocations of memory in each invocation. Our optimized sequential baseline is based on an array of characters of a predefined size computed to ensure the representation always fits in it: 11 characters in our case, including a 0 value to mark the end of the string. The application requires a single array of this kind, as it is reused for the hashing of each different input item. Also, this string was filled in with translations of the values to hash which were computed in a suboptimal way. Concretely, the initial code used conditional statements to translate from single digit integers in which the input was decomposed to the associated character in their string representation; for example, the value associated with 1 would be the character "1" and so on. This was replaced with straight computations, as since it is standard that the ASCII characters from "0" to "9" start at the value 48 for the character "0" and are consecutively mapped, it follows that the ASCII character for the single digit integer

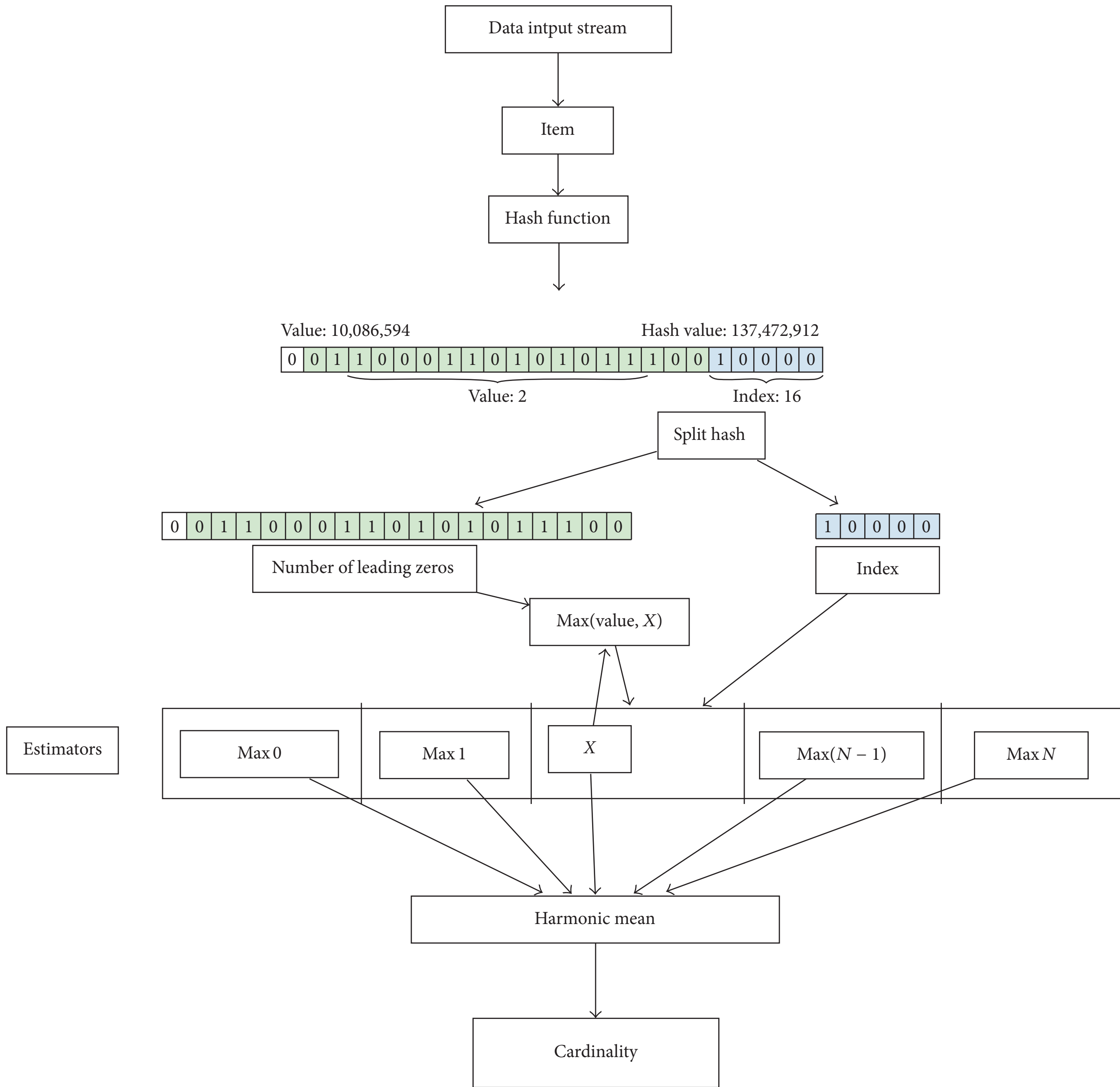


Figure 1: Sequential implementation of the HyperLogLog algorithm.

x is always $x + 48$. Both improvements made our sequential baseline one order of magnitude faster (concretely about 16 times) than the initial implementation.

Two parallel versions of the HyperLogLog algorithm were developed, one based on OpenMP and another based on OpenCL. Both implementations follow the strategy of decomposing the input stream in chunks that are processed in sequence, the parallelization being applied to the loop that processes the items in each chunk. We now describe the two implementations in turn.

3.1. OpenMP Parallelization. This implementation processes the data in chunks whose size can be adjusted. The processing of each chunk is parallelized by statically partitioning the iterations of the loop that processes the chunk among the available threads using the `omp parallel for` compiler directive. This way each thread processes different items in the chunk concurrently, as shown in Figure 2. As we can see in Figure 1, all the steps of the processing of each item can be safely performed in parallel except the update of the associated bucket of the algorithm, as each bucket is shared by many items, and sometimes two or more of these items could be processed simultaneously by different threads. Unfortunately OpenMP does not provide atomic `max` operations in C++, the language in which we developed our application. Also, a coarse-grained strategy such as protecting all the buckets

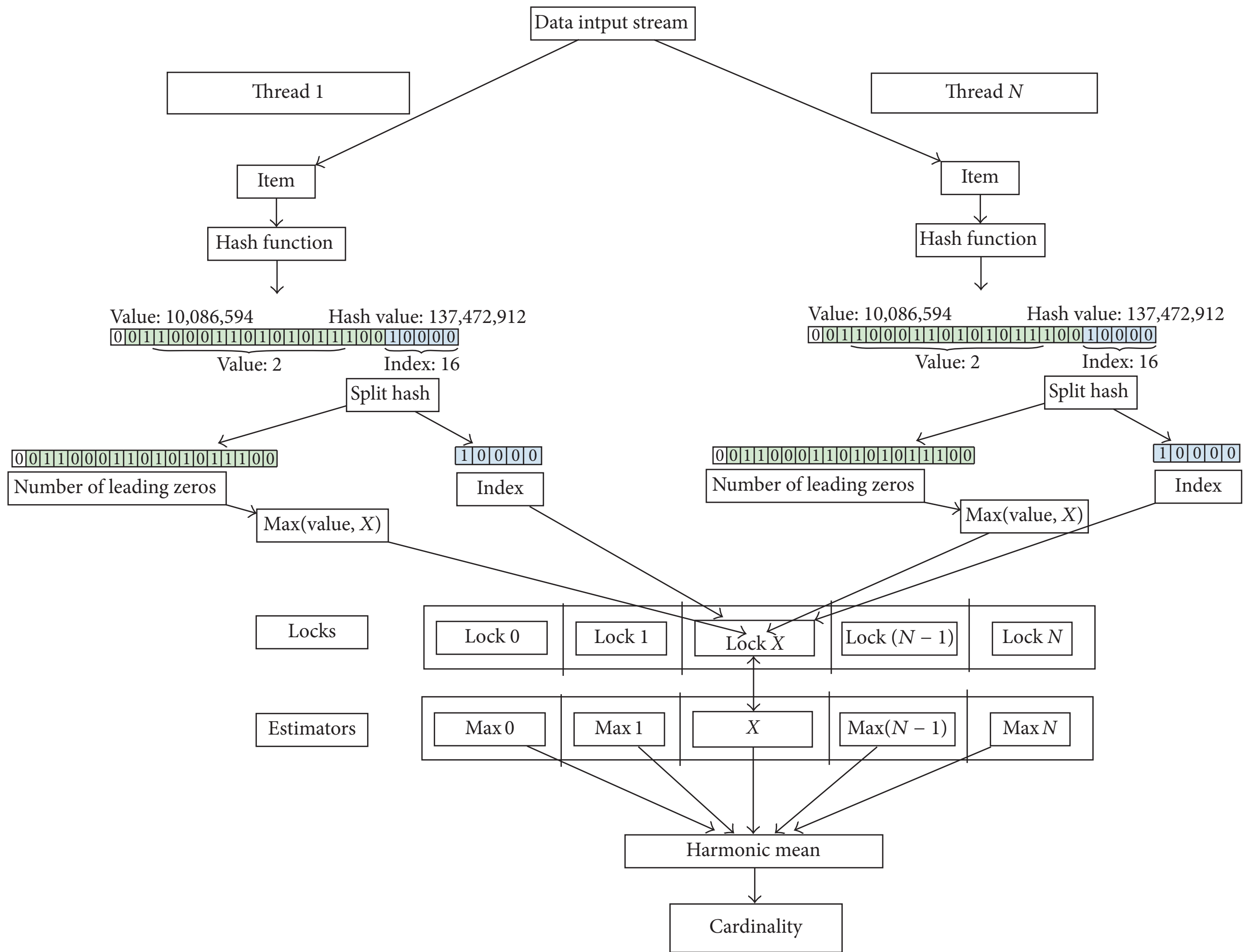


Figure 2: OpenMP implementation of the HyperLogLog algorithm.

with a single lock would heavily serialize the operation of the threads due to the large contention experienced in this lock. For this reason, we ensured the safeness of the update of the buckets by following a fine-grained locking approach in which each bucket is protected by a different lock, which is reflected at the bottom of Figure 2.

Even when contention is much lower using fine-grained locking, locking and unlocking have still some cost. In addition, in executions in systems with many cores such as the Intel Xeon Phi, the contention can be noticeable. For this reason we further optimized this reduction stage by comparing the locally obtained *nleadingZeros* value with the one currently stored in bucket associated with the item being processed (M_{index} in Algorithm 1) and only trying to acquire the lock to update the bucket if the local value is greater, as otherwise the value in the bucket would remain unchanged. Of course after acquiring the lock *nleadingZeros* is compared again to M_{index} to make sure it is still larger in order to decide whether the bucket must actually be updated.

3.2. OpenCL Parallelization. OpenCL is a standard for heterogeneous computing which, contrary to other popular approaches such as CUDA, provides large portability, enabling applications to be run in a wide variety of accelerators. Unfortunately, contrary to OpenMP, OpenCL requires much boilerplate code as well as the management of new concepts such as buffers, command queues, and platforms noticeably obscuring the application [3]. This has led to the development of libraries that simplify the use of this paradigm by automating many of tasks it requires and hiding from the developer several of its related concepts, while usually providing minimal overheads with respect to the straight use of OpenCL. We have developed our parallelization using one of these libraries, namely, the Heterogeneous Programming Library (HPL) [4], which has shown to provide excellent programmability for heterogeneous applications while incurring in negligible overheads. Although HPL supports writing the kernels in OpenCL C [5] our implementation relies on the embedded language introduced in [4], which is translated at runtime into OpenCL C by the library.

Our OpenCL implementation is based on a kernel that parallelizes the innermost loop in Algorithm 1. The kernel executes a different OpenCL work-item, which is basically a parallel thread of execution, for a number of elements in the chunk that the user can configure in order to control the

granularity. This way, the loop is replaced by the execution of the kernel in the selected accelerator. The kernel code chooses the items from the chunk based on the unique identifier of the work-item and then applies the same steps as the sequential implementation, namely, hashing, split of the hash, count of leading zeros, and update of the associated bucket. As in the OpenMP implementation, this last stage poses a problem, as multiple work-items can try to update the same bucket in parallel. Our implementation relies on an `atomic_max` routine provided by OpenCL to avoid the potential races in these updates. In order to reduce ping-pong problems in the caches due to the exclusive ownership required by writes, our implementation always tests whether the locally computed *nleadingZeros* is greater than the current value in the bucket, and only in that case does it attempt to perform the atomic operation. Another optimization we tested consisted in using the OpenCL local memory as a cache of the buckets for each work-group in order to try speedup of the checks and the updates on the buckets. The rationale for this optimization was that (a) the access to the buckets in local memory should be faster than in global memory and (b) since each work-group would have its own copy, there should be less contention in the atomic updates. Nevertheless, this turned out to provide no advantage with respect to the direct usage of a single array of buckets stored in the global memory of the device and shared by all the work-items in execution.

Another difference between this implementation and the OpenMP one is that OpenCL kernels are typically run in an accelerator that has a separate memory. This implies that the data must be transferred between the main host memory and the accelerator memory for its processing. In our case, each chunk must be transferred to the accelerator before its processing, and at the end of the main loop the buckets must be transferred from the accelerator to the host. Since the chunk is the main memory object to transfer and this operation happens repetitively, its size was tuned seeking the best performance for each platform as we will see in Section 4.

4. Experimental Results

In this section we evaluate our implementations in three systems with the same CPU and main memory configuration, but each computer has a different kind of accelerators. Namely, each host was a dual-socket system with two Intel Xeon E5-2660 Sandy Bridge processors with eight 2.2 Ghz cores and 2-way hyperthreading (8×2 threads per processor, for a total of 32) and 64 GB of RAM, offering a single-precision theoretical peak performance of 563 GFLOPS. The accelerators available in the three computers were

(i) an Intel Xeon Phi 5110P with sixty 1.053 GHz cores with 8 GB of RAM; Intel OpenCL driver version 1.24.5.0.8; Single-precision theoretical peak performance of 2022 GFLOPS

(ii) an NVIDIA Tesla K20m with Kepler GPU architecture (2496 cores at 705 MHz) and 5 GB GDDR5; NVIDIA OpenCL driver version 340.58; Single-precision theoretical peak performance of 3524 GFLOPS

(iii) an AMD FirePro S9150 with Hawaii GPU architecture (2816 cores at 900 MHz) and 16 GB GDDR5; AMD OpenCL driver version 1702.3; Single-precision theoretical peak performance of 5070 GFLOPS.

The codes were compiled using the Intel compiler version 13.1.1 enabling the `O3` optimization level. This compiler was chosen both because Intel compilers are the recommended ones for applications that run in the Xeon Phi, and this way the same compiler can be used across all the experiments, and because we observed that it provided better performance in the Sandy Bridge host than the other compiler we had available (g++ 4.9.2). The only exception is the OpenCL kernels, which are compiled at runtime by the JIT compiler of the OpenCL environment, as is the usual practice. In the experiments we measured the time required by the execution of the HLL algorithm itself; thus the measurement does not include the load of the items to process in the host memory. The measurements include however the time required to compile the kernels and transfer the data between the host memory and the accelerator when running in OpenCL. In order to eliminate outlier results, the times reported are the average of 10 runs for each experiment.

The experiments have been performed using 256 buckets for the HLL algorithm and a data set of 10^9 items, each item being an integer of 32 bits, thus totaling an input of 4 GB. In order to establish whether the results were representative for other problem sizes, a study on the standard deviation and the evolution of the runtime of the processing of each new chunk as the problem size grows was performed using all our implementations and platforms. In all the cases we observed that, after the processing of the first chunk, the processing time per chunk always got stable and regular, meaning that the average processing time of each new chunk remained constant. Also, the deviations observed after this short initialization period were small. Namely, the standard deviation of the chunks processing time varied between a minimum of 0.16% of the average chunk processing time for our optimized sequential baseline executed in the Xeon E5-2660 host and a maximum of 8.26% for the HPL/OpenCL implementation in the AMD FirePro S9150, which presents the shortest runtimes. The larger variability obtained for the first chunk was to be expected, as not only are caches heated and do conflicts in the buckets happen more frequently at the beginning, but, in the case of the OpenCL based codes, the buffer allocations in the devices and the compilation of the kernels only happen for the very first chunk. As a result, the relative performance observed in our experiments is representative of the ones for other large problem sizes that justify resorting to an approximate algorithm such as HLL.

We now evaluate the absolute performance and the scalability of the OpenMP implementation in the Sandy Bridge host and the Intel Xeon Phi. This accelerator contains 60 cores, each one of them capable of executing four threads, but since one of the cores is reserved for the system OS, this leaves users with 59 usable cores and thus $59 \times 4 = 236$ parallel threads to execute their applications. Since under OpenMP the Xeon Phi can execute full applications that perform every sort of activity, including I/O, these runs were

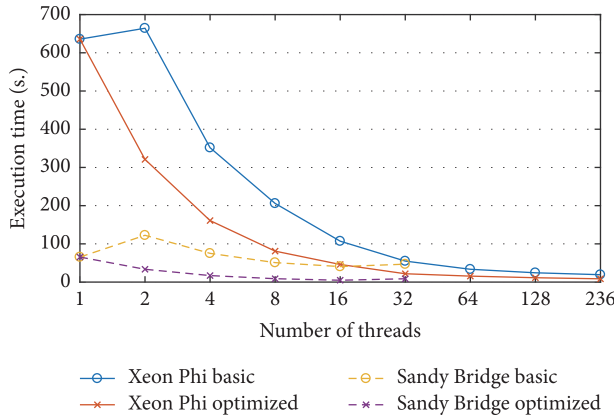


FIGURE 3: OpenMP execution times.

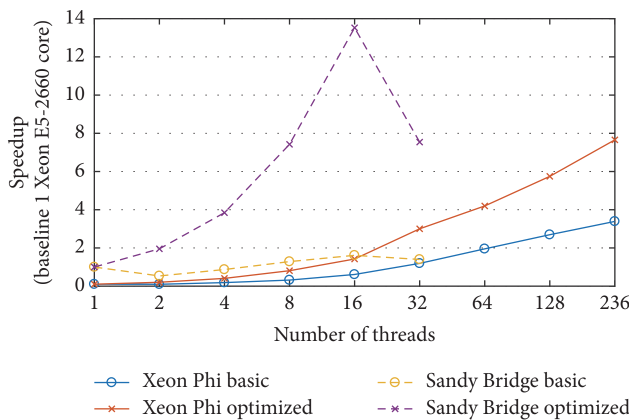


FIGURE 4: OpenMP speedups with respect to the sequential execution in the Intel Xeon E5-2660 CPU.

directly executed in the Xeon Phi and thus they do not involve neither the host nor data transfers between the host and the accelerator. Given a series of tests, the chunk size chosen for this implementation for both systems was 100000 elements, as increases in the chunk size until this point led to improvements in the runtime that plateaued at this point.

Figure 3 shows the absolute execution time of the HLL algorithm in the host and the Xeon Phi when implemented using OpenMP when using from 1 to the maximum number of threads supported in each system. For each platform we show the runtime obtained for our initial version, which always locked the access to the buckets, and the optimized one that reduces the amount of locking required. The associated speedups with respect to the serial optimized execution in one Xeon E5-2660 CPU core are reflected in Figure 4. As we can see, the optimization applied is very important for both systems. The optimized version scales linearly in the Sandy Bridge host, for which using hyperthreading is counterproductive, as the increased contention in the locks outweighs the additional parallelism exploited. Regarding the Xeon Phi, its behavior scales very linearly up to 32 threads, as in this region each duplication of the number of threads leads to a growth of the speedup obtained between 87% and 100%. After that point the collisions in the locks of the 256 buckets begin to happen more often, leading to increases of the speedup between 33% and 54%. By using all the hardware threads, the optimized implementation reaches a 78 times' speedup compared to a single Xeon Phi core. Unfortunately, as Figure 3 shows, the sequential version of the program runs about ten times slower in a core of the Xeon Phi than in a core of the Sandy Bridge. An important reason for this is that while vectorization is known to be critical to achieve good performance in this platform, unfortunately the two most important loops in this application, which account for 84% of its runtime, have a variable number of iterations and present inherent dependence between consecutive iterations of the loops, which precludes their vectorization. For this reason, as shown in Figure 4, the speedup of the Xeon Phi is only 7.7 with respect to the sequential execution in the Xeon E5-2660 CPU. As a result, the Sandy Bridge host, which reaches a speedup of 13.5, clearly outperforms the Xeon Phi in this problem.

As for our HPL/OpenCL implementation, it was tested in the three accelerators we had available as well as in the host. This was possible thanks to the portability that this standard provides. Still, it is important to notice that functional portability does not imply performance portability. In fact the search for strategies to develop performance-portable codes on top of OpenCL is an important research topic [6, 7]. As a result, we tuned the code for each platform seeking the best chunk size, work-group size, and number of items to process in each work-item. For the first parameter we tried all the possible chunk sizes that were a power of 2, while the two last parameters were found using an exhaustive search limited by the maximum work-group size supported by each platform and the number of elements in each chunk. The runtimes found for the neighbor chunk sizes of the optimal one found for each platform, which had a size of one-half and double of it, were very similar, thus indicating that a more fine-grained search in the neighborhood of the best chunk size found would be of little use. The optimum values found, shown in Table 1, prove indeed the different requirements and characteristics of these platforms.

Figure 5 summarizes the speedups achieved by all the versions we developed, where in the case of HPL the optimized version checks the value of the bucket to decide whether to apply the atomic `max` operation, while the unoptimized version always applies this atomic operation for every item processed. We can see that this technique was not interesting in OpenCL, as it slightly reduced the performance in the two GPUs and had a negligible impact in the Intel platforms. Another interesting conclusion is that the high parallelism of GPUs allows them to clearly outperform the other platforms in this problem, as they reached maximum speedups of 44.7 (K20) and 88.6 (S9150), noticeable better than the 14.4 of the Xeon Phi and the 16.7 of the Sandy Bridge. As for the comparison of OpenCL/HPL with OpenMP, the former one performs better than the latter one, the maximum speedups for OpenMP being 7.7 in the Xeon Phi and 13.5 in the Sandy Bridge. We must take into account the fact that when using OpenCL, the binary of the kernel is generated by the OpenCL driver, and it may be the case that it generates more efficient code for this concrete problem than the Intel compiler. We

Table 1: Optimum parameters for each platform.

Platform	Xeon Phi	Sandy Bridge	K20	S9150
Chunk size (in items)	2^{26}	2^{21}	2^{26}	2^{27}
Work-group size	256	1024	128	128
Elements per work-item	1	1	2	4

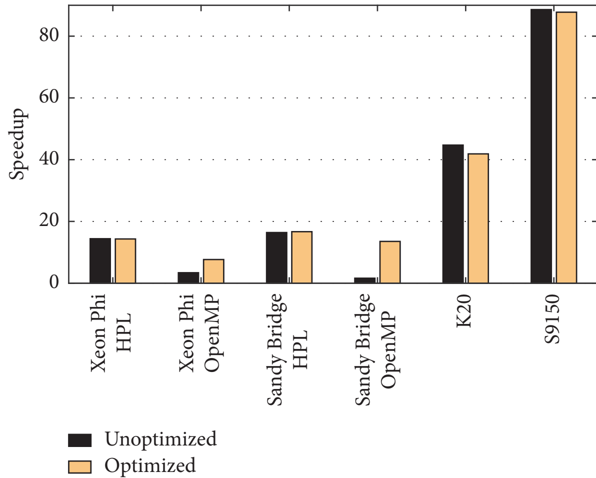


Figure 5: Speedups achieved by all the parallel versions with respect to the sequential execution in the Intel Xeon E5-2660 CPU.

have also looked into the possibility that the usage of the atomic operation provided by OpenCL instead of the more expensive lock-based implementation required by OpenMP was another reason for the improved performance. Profiling however showed that the function where locking/unlocking happens only accounts for 3.6% of the runtime of the OpenMP code in the Xeon Phi, which is the system where the locks can suffer more contention. As a result this second hypothesis has been discarded.

5. Related Work

The field of estimation algorithms is not new. MinCount [8, 9] is an estimation algorithm that helps estimate cardinality with $O(\epsilon - 2\log n)$ space requirements, its accuracy being $1/\sqrt{m}$, where m is the maximum number of hash values. After MinCount, researchers continued to propose new algorithms in order to save space and increase both speed and accuracy. This way, the LogLog algorithm [10] provided a big leap of intuition in the field of cardinality estimation. This algorithm estimates the cardinality of a data set by using probability statistics and basically estimates cardinality on the probability of a number occurring. To be more accurate the algorithm does this a number of times and then takes the geometric mean of the values. The effectiveness of this algorithm and others on different data types was evaluated in [11].

Later, HyperLogLog [2] largely increased the accuracy of the original LogLog proposal by using a harmonic mean [12] for value correction instead of a geometric mean [13]. The algorithm has become popular in the growing field of computer science algorithms called big data because of its excellent properties for the processing of vast amounts of information, there being specialized implementations for important big data problems such as DNA processing with parallel implementations for shared memory based on OpenMP [14] and distributed memory based on MPI [15]. A generic parallel implementation of HLL based on Parallel Java Map Reduce has also been proposed [16]. The report provides very few details, and the runtimes reported, which use up to 8 cores, are much higher than those of our implementations for the same numbers of cores. Since it is a C++ code parallelized with OpenMP, [14] is the most related to our approach. However it relies on C++ strings and complex flows with switch statements, whose removal accelerated by 16 times our sequential implementation. Also, as we have seen in the previous section that OpenCL achieved better performance than OpenMP in our CPU and Xeon Phi. Finally, since we did not find any implementation of HLL for accelerators in the literature, our portable OpenCL based proposal and the parameterization we performed are of particular interest.

A more recent version of the algorithm is HyperLogLog++ (HLL++) [17], which increases the power of the already very efficient HyperLogLog algorithm. One of the changes is that instead of 32-bit hashes HLL++ uses 64-bit hashes, so there is no longer a need for long range correction. Also the authors of HLL++ found through trial and error a value correction table that makes the algorithm even more accurate. Finally, in [18] there is an algorithm that is quite similar to HyperLogLog, but instead of a harmonic mean it uses an inverse of an arithmetic mean as normalizing function. Neither of these variations of HLL imply modifications in the computationally intensive part of the algorithm, the changes involving only the final correction stage, which runs in a totally negligible time. This way, both our parallelization schemes for HLL as well as their impact on runtime would be totally identical for these algorithms.

6. Conclusions

The increasingly widespread management of large amounts of data is a critical challenge that demands both algorithms suited to the particular needs of these problems and optimized implementations of them. One of the strategies applicable to some of these problems is approximate computing, which provides sufficiently good results rather than accurate ones, thus saving considerable computational resources while allowing solving the problem at hand. The HyperLogLog algorithm for the count-distinct problem belonging to this family of solutions has received considerable attention thanks to its reduced memory requirements. In this paper we have explored several parallel implementations of this algorithm

in order to further facilitate its application on (very) large sets of data. Namely, we developed portable implementations based on OpenMP and OpenCL which were evaluated in four platforms. Our OpenCL implementations allow exploiting the high performance provided by GPUs, where we obtained speedups of up to 88.6 with respect to an optimized sequential implementation. When these accelerators are not available, the OpenCL implementation is still useful, as it can run on regular CPUs, achieving a speedup of 16.7 in a Sandy Bridge, outperforming the maximum speedup of 13.5 obtained by our OpenMP implementation.

As future work we contemplate extending our parallel implementations to exploit multiple accelerators attached to the same computer using the advanced features of HPL [19], as well as the development of distributed memory versions that parallelize the algorithm in homogeneous and heterogeneous clusters.

Conflicts of Interest

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

Acknowledgments

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (Projects TIN2013-42148-P and TIN2016-75845-P) as well as by the Xunta de Galicia (Centro Singular de Investigación de Galicia accreditation 2016–2019) and the European Union (European Regional Development Fund, ERDF) under Grant Ref. ED431G/01.

References

[1] M. Mitzenmacher, "Compressed Bloom filters," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604–612, 2002.

[2] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *Proceedings of the Conference on Analysis of Algorithms (AofA '07)*, P. Jacquet, Ed., pp. 127–145, Juan des Pins, France, June 2007.

[3] R. V. Van Nieuwpoort and J. W. Romein, "Correlating radio astronomy signals with many-core hardware," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 88–114, 2011.

[4] M. Viñas, Z. Bozkus, and B. B. Fraguela, "Exploiting heterogeneous parallelism with the Heterogeneous Programming Library," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1627–1638, 2013.

[5] M. Viñas, B. B. Fraguela, Z. Bozkus, and D. Andrade, "Improving OpenCL programmability with the Heterogeneous Programming Library," *Procedia Computer Science*, vol. 51, pp. 110–119, 2015.

[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[7] J. F. Fabeiro, D. Andrade, and B. B. Fraguela, "Writing a performance-portable matrix multiplication," *Parallel Computing*, vol. 52, pp. 65–77, 2016.

[8] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '02)*, vol. 2483, pp. 1–10, Springer Berlin Heidelberg, Cambridge, Mass, USA, September 2002.

[9] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[10] M. Durand and P. Flajolet, "Loglog Counting of Large Cardinalities," in *Proceedings of the 11th Annual European Symposium (ESA '03)*, vol. 2832 of *Lecture Notes in Computer Science*, pp. 605–617, Springer Berlin Heidelberg, Budapest, Hungary, September 2003.

[11] K. Aouiche and D. Lemire, "A comparison of five probabilistic view-size estimation techniques in OLAP," in *Proceedings of the 10th ACM International Workshop on Data Warehousing and OLAP (DOLAP '07)*, pp. 17–24, Lisbon, Portugal, November 2007.

[12] Y.-M. Chu and W.-F. Xia, "Two sharp inequalities for power mean, geometric mean, and harmonic mean," *Journal of Inequalities and Applications*, vol. 2009, Article ID 741923, 2009.

[13] D. Mindlin, "On the Relationship between arithmetic and geometric returns," *SSRN Electronic Journal*, 2012.

[14] L. C. Irber Jr. and C. T. Brown, "Efficient cardinality estimation for k-mers in large DNA sequencing data sets," *bioRxiv*, 2016.

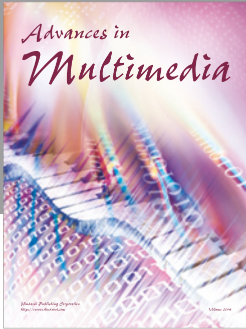
[15] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de Bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, pp. 437–448, New Orleans, La, USA, November 2014.

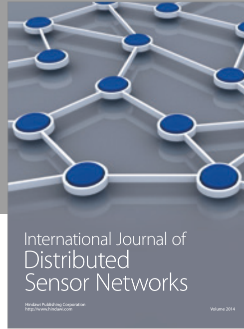
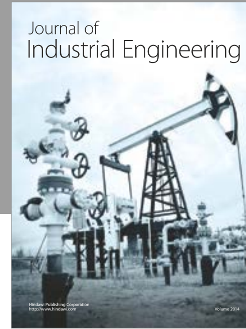
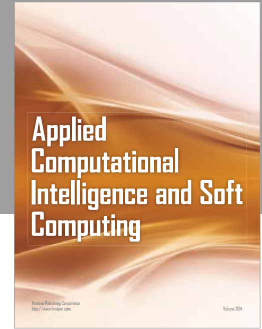
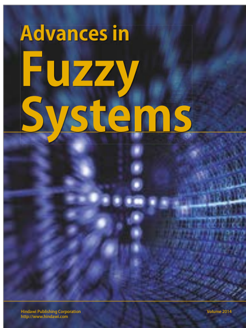
[16] K. Kumar and S. Subash, "Approximate large multiset cardinality using map reduce," Tech. Rep., Rochester Institute Of Technology, 2015.

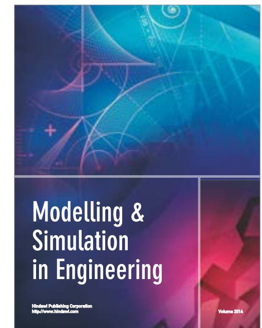
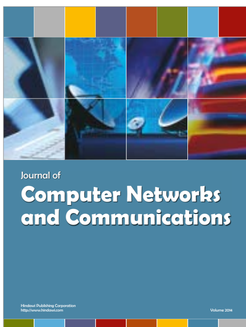
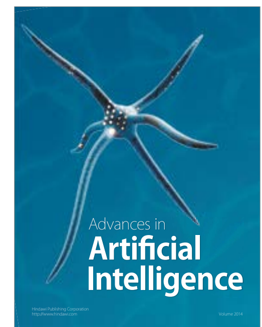
[17] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*, pp. 683–692, Genoa, Italy, March 2013.

[18] J. Lumbroso, "An optimal cardinality estimation algorithm based on order statistics and its full analysis," in *Proceedings of the 21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA '10)*, Michael Drmota and Bernhard Gittenberger, Ed., pp. 489–504, Institute of Discrete Mathematics and Geometry, Vienna, Austria, July 2010.

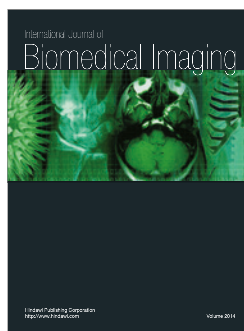
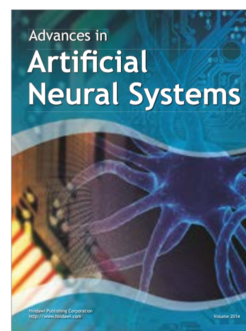
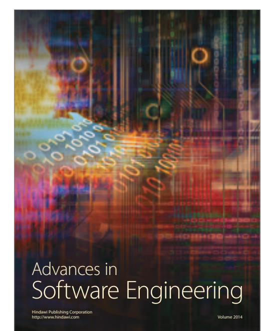
[19] M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo, "High productivity multi-device exploitation with the Heterogeneous Programming Library," *Journal of Parallel and Distributed Computing*, vol. 101, pp. 51–68, 2017.

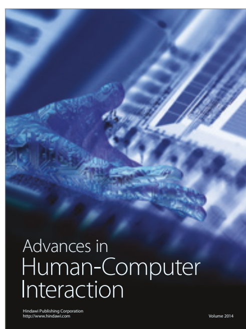
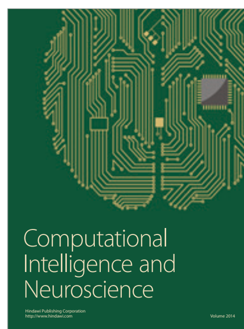

Advances in
Multimedia

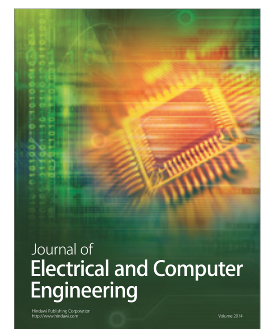
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
https://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering