International Journal of Innovative Computing, Information and Control
Volume **9**, Number **12**, December **2013**

ICIC International ©2013 ISSN 1349-4198
pp. 4825–4849

DIGITAL HAMMING WEIGHT AND DISTANCE ANALYZERS FOR BINARY VECTORS AND MATRICES

Valery Sklyarov and Iouliia Skliarova

Institute of Electronics and Telematics Engineering of Aveiro
Department of Electronics, Telecommunications and Informatics
University of Aveiro
3810-193 Aveiro, Portugal
{ skl; iouliia }@ua.pt

Received December 2012; revised May 2013

Abstract. *In this paper we explore modules that can analyze binary vectors and matrices and efficiently solve a wide range of problems that involve the computation of Hamming weights or Hamming distances, producing counts and/or comparisons of these, as well as sorting and searching. A set of designs for such modules is proposed and all the designs have been evaluated both theoretically and practically. The practical evaluation involved numerous experiments with hardware implementations using the most recent extensible processing platform that incorporates reconfigurable logic. The objective was to achieve high performance within reasonable resources. As a result, novel solutions for Hamming weight counters/comparators have been identified that have better cost and latency than the best known alternatives. Network-based sorters and searchers with reusable cores are also discussed and these enable high throughput to be achieved with relatively modest resources. The paper shows that similar results cannot be obtained using the best known and most frequently used even-odd merge and bitonic merge networks. Finally, a complete architecture for an analyzer is presented, part of which (covering the modules indicated above) has been completely implemented and prototyped in hardware.*
Keywords: Combinatorial search, Data/signal/image processing, Hamming weight, Parallel systems, Performance/resources analysis, Pipeline, Reconfigurable hardware, Sort/search

1. Introduction.

The *Hamming weight* $w(A)$ of a binary vector $A = \{a_0, \ldots, a_{N-1}\}$ is the number of one bits in the vector, which ranges from 0 to N [1]. The *Hamming distance* $d(A, B)$ between two vectors A and B is the number of corresponding elements that differ. Certain applications that are fundamental to information and computer science require $w(A)$ and $d(A, B)$ to be calculated and analyzed for either a single vector, or a set of vectors that are the rows/columns of a binary matrix. Such applications can be found in digital signal processing [2], image [3] and data processing [4], encoding and error correction [5], cryptography [6], combinatorial search [7], DNA computing [8] and many other areas.

For example, $w(A)$ often needs to be compared with a fixed *threshold* κ, or with $w(B)$, where $B = \{b_0, \ldots, b_{Q-1}\}$ is another binary vector and Q and N may or may not be equal. Examples of applications where this can be the case include digital filtering [9,10], piecewise multivariate linear functions [11], pattern matching/recognition [12,13], problem solving in Boolean space [14], combinatorial search [15,16], and encoding for data compression [17]. Many of these require Hamming weight comparators with very high throughput. Streaming applications that are frequently used receive vectors sequentially,

and the allowable delay between receiving a vector on inputs and outputting a result is limited. Thus, increased speed is always a major requirement. Many research (e.g., [18-20]) and application problems (e.g., [21,22]) necessitate not only counting and/or comparing Hamming weights, but also an analysis of the distribution of weights in a matrix (or in a stream). We might be interested in answering such questions as: what is the maximum and/or minimum weight in a set of vectors? how many vectors are there with the same weight (in succession or in the entire sequence)? what is the sorted set of vectors? Since any collection of vectors in a stream can be modeled by a binary matrix, we can answer the questions above by implementing real-time processing of such matrices in a way that provides for the following:

1. The weight of each incoming vector is: a) determined with the minimal achievable delay; and b) temporarily stored in memory for further processing.
2. Various types of parallel processing can be applied to the collected weights, such as the run-time computation of minimum/maximum values, sorting the values, and counting the frequency of occurrence of items.
3. Since the size of temporary storage is limited, it can be organized as a FIFO (first in first out). As soon as the memory is full, the vectors are shifted from its inputs to the outputs. It may be difficult to keep maximum/minimum values in a dedicated register and to compare them sequentially with each incoming vector. As soon as vectors are extracted from the FIFO, we need to provide for the rapid update of the analysis results for data that are currently kept in memory. This task in particular is important for priority buffers/queues [23].

Let us consider another type of application, from combinatorial search. Many optimization algorithms in this context reduce to the covering problem [14]. Let a set $\Theta = \{\theta_0, \ldots, \theta_{E-1}\}$ and its subsets $\Phi_0, \ldots, \Phi_{U-1}$, for which $\Phi_0 \cup \ldots \cup \Phi_{U-1} = \Theta$ be given. The shortest (minimum) covering for the set Θ is defined as the smallest number of subsets for which their union is equal to Θ. In [14] the set $\{\Phi_0, \ldots, \Phi_{U-1}\}$ is represented by a binary matrix, the U rows of which correspond to the subsets of the given set and the E columns to the elements of the set Θ. A value 1 in a row u $(0 \leq u < U)$ and column e $(0 \leq e < E)$ indicates that the row u covers the column e. The covering problem is solved if we can find the minimum number of rows in which there is at least one value 1 in each column. It is known [16] that this problem involves numerous procedures counting the Hamming weights in the binary vectors that represent the rows and columns of the matrix, ordering the rows and columns and applying some task-specific operations. Such problems are very large and time consuming [24] so acceleration is greatly required.

Computing Hamming distances is needed in applications that use Hamming codes (see, for example, [25,26]). However, since $d(A, B) = w(A\ XOR\ B)$ we can find Hamming distances as the Hamming weights of "XORed" arguments A and B. The Hamming weight for a general vector (not necessarily binary) is defined as the number of non-zero elements. Thus, all the methods outlined above can be applied to any type of vector, such as [8], and the only difference is an initial comparison-based operation that enables the input to be presented in the form of a binary vector. The next section will discuss this in detail.

This paper is dedicated to fast parallel application-specific systems that are targeted to FPGAs (Field-Programmable Gate Arrays) and provide for the following:

1. Finding the Hamming weight of a binary vector, or the Hamming weights of a set of binary vectors that are the rows/columns of a matrix, faster and cheaper (i.e., with less resources) than the best known previously published designs. Clearly, Hamming distances for binary vectors can also be computed easily.

2. Run-time processing of Hamming weights (such as searching, sorting and determining the frequency of repetitions) faster and cheaper than the best known previously published designs.

The remainder of this paper contains 8 sections. Section 2 analyzes the known methods and designs and suggests ways for improvements. Section 3 is dedicated to identifying potential practical applications of the results and gives a number of real-world examples from different areas. Section 4 describes the look-up table based circuits we are proposing. Section 5 is dedicated to the counting networks that are suggested, which provide minimal delay in combinational logic for pipelined implementations. Section 6 covers sorting networks with reusable cores. Section 7 presents the results of experiments and comparisons. Section 8 describes the final architecture of the analyzer and potential applications. The conclusion is given in Section 9.

2. **Related Work.** The majority of the known methods and designs that are relevant to this paper are in four main areas: 1) parallel counters (e.g., [1,27,28]); 2) networks for searching, sorting and counting (e.g., [4,29-35]); 3) highly parallel and eventually pipelined designs (e.g., [4,31]); and 4) rational combination of sequential and parallel operations through the reuse of processing cores (e.g., [4,32]). In Section 2 we will analyze the published results in these areas and suggest the ways for potential improvements.

State-of-the-art Hamming weight comparators (HWC) have been exhaustively studied in [1]. The charts presented (Figure 8 in [1]) compare the cost (i.e., the number of gates) and the latency (i.e., the number of gate levels) for three selected methods, those of Pedroni [9], Piestrak [35] and Parhami [1]. It is argued that the cost of HWC from [1] is the best for all values of N (N is the size of the given vector) while the latency up to $N = 64$ is better for the method [35]. For $N > 64$ the method [1] again is claimed to be the best. A thorough analysis of the known HWCs reported in publications permits us to conclude the following: the existing methods mainly involve *parallel counters* (i.e., circuits that execute combinational counting of Hamming weights for given binary vectors) and *sorting networks* or their varieties, such as [9,36]. Note that the majority of HWCs are based mainly on circuits that calculate the Hamming weights of individual vectors. Clearly, the latter can be organized in matrices for streaming applications. Table 1 presents expressions for cost and latency from [1] for fixed threshold and two-vector Hamming weight comparators, where κ is the threshold, γ_{FA} is the cost of a full-adder (FA) relative to a gate, δ_{sum} is the delay of an FA, and δ_{carry} is the delay of carry propagation in the FA.

We found that although the results of the comparison [1] are correct for the alternatives discussed, they are not absolutely justified. For example, the number of gate levels of the circuit [1] is given by the expression $(\log_2 N - 1) \times (\delta_{sum} + \delta_{carry}) + 1$, where the last '1' in this expression is the delay of the carry network (Figure 4 in [1]). Any stage of

TABLE 1. Cost and latency expressions for HWCs from [1]

Design	*Fixed-threshold HWC*		*Two-vector HWC*	
	Cost	*Latency*	*Cost*	*Latency*
Pedroni [9,36]	$2 \times \kappa \times N$	$N + \kappa - 1$	$N \times (N + 1)$	$2 \times N$
Piestrak [35]	$N \times (\log_2 N)^2/2$	$\log_2 N(\log_2 N + 1)/2$	$N \times (\log_2 N)^2 + N + \log_2 N$	$(\log_2 N + 1) \times (\log_2 N + 1)/2$
Parhami [1]	$(N - \log_2 N - 1) \times \gamma_{FA} + \log_2 N$	$(\log_2 N - 1) \times (\delta_{sum} + \delta_{carry}) + 1$	$2 \times (N - \log_2 N - 1) \times \gamma_{FA} + 4 \times \log_2 N$	$(\log_2 N - 1) \times (\delta_{sum} + \delta_{carry}) + 1$

this network includes 3 gate levels and the carry signal has to propagate through all the stages sequentially. Thus, the delay '1' is questionable. There are also doubts about the cost. It is assumed in [1] that a standard full-adder (FA) needs 9 gates, which is correct for ASICs. However, if we are talking about ASICs, then perhaps any gate is *NAND*. Thus, other components of the circuits, such as *OR* and *AND* (including the carry network) have to be counted in *NAND* too, which was not done. An analysis of recent publications clearly demonstrates that the most frequently exploited devices for Hamming weight/distance computation and comparison and the reordering of these are FPGAs. However, if we are using FPGAs, what is the significance of the minimum gate count? Indeed, the criteria for design quality are quite different. This paper suggests two new designs for Hamming weight/distance computation and comparison. The first design is based on look-up tables (LUTs) and it provides better cost and latency than the best previously published alternatives (see Table 1). The second design involves the proposed counting networks that are very regular, easily scalable and can be pipelined with negligible delays between pipeline registers. They also fit well to embedded blocks such as digital signal processing (DSP) slices that are widely available in recent FPGAs.

The second problem that is described in this paper is *sorting*, *searching* and *item frequency computation* of previously calculated and saved Hamming weights. Once again performance is a very important feature. The fastest known parallel sorting methods are based on the *even-odd merge* and *bitonic merge* networks [17,29-32]. The depth $D(N)$ of a network, that sorts N data items, is the minimal number of data dependent steps $S_0, \ldots, S_k, \ldots, S_{D(N)-1}$ that have to be executed sequentially. This characteristic is important for both pure combinational and sequential implementations. In the first case, circuits operating at any step k use the results of circuits from the previous step $k-1$; i.e., there exists a data dependency for step k on step $k-1$. If we assume that the propagation delays of all steps are equal, then the total delay is proportional to $D(N)$. For sequential implementations, $D(N)$ determines the number of consecutively executed steps (clock cycles) and ultimately the throughput of the network. If N is a power p of 2 (i.e., $N = 2^p$) then $D(N = 2^p) = p \times (p+1)/2$ [17,29,31] for both types of networks indicated above. Thus, these networks are very fast. Indeed, sorting 134 million data items ($N = 2^{27}$) can be done with just 378 steps ($D(N = 2^{27}) = 378$). However, there is another problem. The hardware resources required are enormous. Let us analyze the *even-odd merging* network (which is less resource consuming than the *bitonic merging* network). The number of comparators for this network is $C(N = 2^p) = (p^2 - p + 4) \times 2^{p-2} - 1$ [17,29,31] (for the *bitonic* network the number of comparators is a bit larger: $C(N = 2^p) = (p^2 + p) \times 2^{p-2}$). Thus, sorting 134 million items requires 23 689 428 991 comparators. We target our results to FPGA because they can be seen more and more as a universal platform incorporating many complex components that were used autonomously not so long ago. For example, the extensible processing platform (EPP) Xilinx *Zynq* [37] includes dual ARM® Cortex^{TM}-A9 MPCore^{TM} and Artix/Kintex FPGA on the same microchip. A similar platform was introduced by Altera. We found that one comparator for 32-bit data items ($M = 32$) consumes between 17 and 55 Xilinx FPGA slices. To our knowledge the largest existing FPGA XC7V2000T (with 6.8 billion transistors) contains 305 400 slices. Hence, to implement a combinational network-based sorter for 134 million items, we would need about 2.7 million of the most advanced FPGAs. Similar problems arise with other types of implementations, such as those using graphics processing units (GPUs) [30]. Partially sequential implementations permit hardware resources to be reduced but cause a new problem: the number of memory transactions becomes very large [30]. So, 378 steps is a good characteristic but either it is not realizable in practice, or each step becomes very time consuming due to the necessity to split it into numerous sub-steps that require

an excessive number of memory transactions. We suggest in this paper a good compromise between pure combinational parallel operations and stages that sequentially reuse the parallel operations. The emphasis is on circuits that are as regular as possible and thus easily scalable, avoiding redirecting data streams based on multiplexing operations. This is because any multiplexer involves additional propagation delays and complicates interconnections and the latter may cause further additional propagation delays.

Thus, two ways that allow the known results to be improved are going to be explored: 1) discovering regular, easily scalable and simply pipelined designs for Hamming weight/distance counters/comparators based on look-up tables and the counting networks that are proposed; 2) combining network-based reusable combinational and feedback sequential operations with the primary objective of achieving the minimum gate level delays in sorters and searchers. We mentioned that a Hamming distance counter/comparator can be built as a Hamming weight counter/comparator of "*XOR*ed" vectors A and B. Thus except for a set of *XOR* gates, no additional logic is required.

All the proposed designs will be evaluated and compared with the existing alternatives both theoretically and through FPGA prototyping. The experiments will be conducted using the *ZedBoard* [37] with the Xilinx *Zynq xc7z020* microchip incorporating the most recent 7 series FPGA.

3. Practical Applications.

Let us now discuss systems for which high throughput is very important. Many electronic, environmental, medical, and biological applications need to process data streams produced by sensors and measure external parameters within given upper and lower bounds (thresholds). Examples of such measurements include monitoring thermal radiation from volcanic products [38] and digital filtering [2]. Let us describe the problem in the manner shown in Figure 1(a), where sensors $S_0, \ldots, S_{N-1}$ (the value N can be thousands) measure and output data.

A set of data values (SDV) collected at the same time is presented in the form of subsets that include: 1) values that are below the lower bound (SDV$_\downarrow$), 2) values that are between the upper and the lower bounds (SDV$_\leftrightarrow$), and 3) values that are above the upper bound (SDV$_\uparrow$). The number of subsets can be just two (i.e., above the given threshold SDV$_\uparrow$ and below the given threshold SDV$_\downarrow$) or more than three (i.e., there are more than three intervals in which we would like to analyze the data values produced by the sensors).

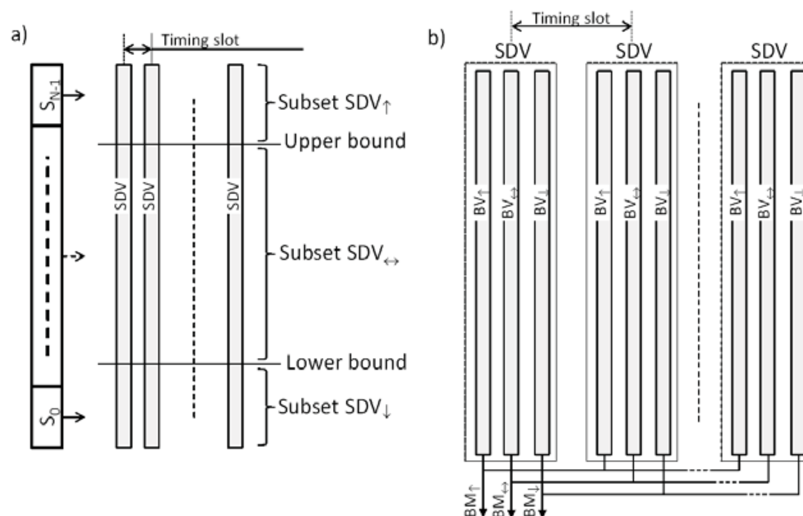

FIGURE 1. Data streams formed by sensors for different types of measurements (a), potential way to digitize and analyze the measured values (b)

Depending on the application, SDVs may be generated with frequencies that are very high (measured in megahertz). Thus, processing such SDVs has to be very fast and high-speed accelerators are essential. Let us look at the model depicted in Figure 1(b), where the entire interval of potential values is digitized. If a measured value falls within a predefined discrete interval, a flag '1' is recorded in the corresponding N-bit binary vector, which contains only zeros initially (see Figure 1(b)). Thus, for the three subsets exemplified above ($SDV_{\downarrow}$, $SDV_{\leftrightarrow}$ and $SDV_{\uparrow}$), three binary vectors ($BV_{\downarrow}$, $BV_{\leftrightarrow}$ and $BV_{\uparrow}$) are built. Sequentially generated SDVs are *SDV streams* and in our model they are represented by three binary matrices ($BM_{\downarrow}$, $BM_{\leftrightarrow}$ and $BM_{\uparrow}$) composed of the vectors $BV_{\downarrow}$, $BV_{\leftrightarrow}$ and $BV_{\uparrow}$ respectively. As we noted above, there can be more or less than three of such matrices.

In many practical cases we would like to analyze the distribution of potential values between different matrices (see Figure 1(b)). For example, we may want to know how often volcanic activity [38] falls to a set of critical values (such as $BM_{\uparrow}$), or how the results of measurements in different environmental, medical, or biological experiments are distributed, or how signals can be filtered in digital and signal processing, and so on. Thus, we need to know how many measured values fall within pre-selected bounds (i.e., what is the Hamming weight of binary matrices such as $BM_{\downarrow}$, $BM_{\leftrightarrow}$ and $BM_{\uparrow}$).

The Hamming weight of each matrix in Figure 1(b) (which is the Hamming weight of all elements in the matrix) indicates its power (the intensity of the relevant signals). Generally, the greater the intensity, the more critical the subset is. The more subsets there are that have critical values, the higher the probability of an event which might happen. Analyzing the Hamming weights in subsequent time slots or in the associated sub-matrices permits the determination of when or where an activity of the measured values is higher or lower. Measuring Hamming distances enables us to check the number of repetitions of activities within chosen subsequent time slots or within certain times. By discovering the maximum and minimum values you can determine when the activity is the highest or the lowest. Sorting the values enables charts showing activities during a chosen time period to be built.

Filtering is the kind of processing shown in Figure 2. Suppose that we are only interested in SDVs with Hamming weights above (or alternatively below) a given threshold κ and we would like to choose just these values for further processing. In this case we need the digital filter shown in Figure 2.

The processing described above (see Figures 1 and 2) requires high performance computations. Indeed, the Hamming weights of very long binary vectors have to be found as fast as possible. Thus, to achieve this, multiple segments and different bits of a vector need to be processed in parallel.

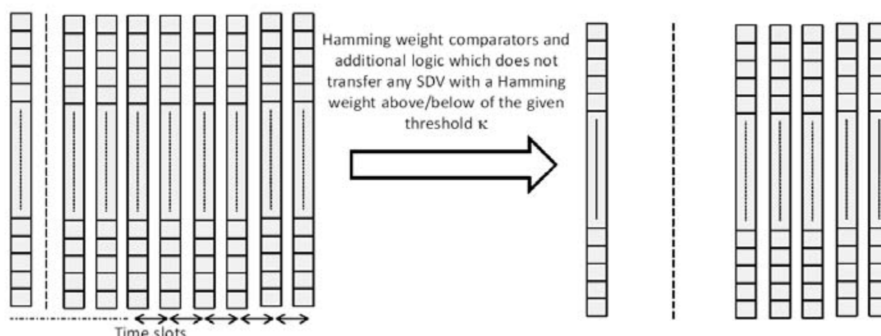

FIGURE 2. An example of digital filtering

Let us explore some other examples where Hamming weight and distance analyzers for binary vectors and matrices can be used efficiently.

Example 3.1. *Suppose there are a number of multiple-valued data items from which we have to choose items with a pre-selected value or, alternatively, we would like to know how many items fall into a pre-selected range of values. For instance, many problems can be solved over ternary vectors that include 3-valued elements: '1', '0' and 'do not care'. The elements are coded somehow by 2-bit binary vectors [39], for example '0' as 00, '1' as 11, and 'do not care' as 01 or 10. In this case the vector 0-1-10, where - is a 'do not care' value, is coded as 00* ***01*** *11* ***10*** *11 00 and the pairs 01 and 10 shown in bold font correspond to 'do not care'. If we need to know the number of 'do not care' values we will use XOR gates for the 2-bit binary vectors and the elements from large vectors can be processed in parallel. The XOR operation applied to the bits in element values 01 and 10 gives 1 so this indicates that the relevant element is 'do not care'; XORing the bits in 00 or 11 results in 0. The number of 1s in the resulting vector is equal to the number of 'do not care' values in the associated ternary vector. Thus the Hamming weight for the resulting vector gives the number of 'do not care' values in the corresponding ternary vector. Similar operations can be applied to binary vectors with elements that are associated with data items falling into a pre-selected interval. Thus, we need Hamming weight computations. The algorithms [14-16,24,39,40] involve such operations over sets of vectors (i.e., over matrices). We will additionally discuss this problem in Section 6 below.*

Example 3.2. *Suppose there are pre-defined values* $\alpha_1, \ldots, \alpha_Z$ *and we would like to discover how many values* $\alpha_z \in \{\alpha_1, \ldots, \alpha_Z\}$ *can be found in a given set. Suppose we have a set of data items* $I_0, \ldots, I_N$*. The result* $R(\alpha_z)$ *of comparing* $\alpha_z \in \{\alpha_1, \ldots, \alpha_Z\}$ *with all the items* $I_0, \ldots, I_N$ *is a binary vector. The Hamming weight of the vector* $R(\alpha_z)$ *is equal to the number of items with the value* α_z*. Sorting the results* $R(\alpha_1), \ldots, R(\alpha_Z)$ *gives the distribution of data items with the values from* $\{\alpha_1, \ldots, \alpha_Z\}$ *in the set* $I_0, \ldots, I_N$*. Such a problem appears in pattern recognition, image and signal processing. Thus, we need Hamming weight computations and sorting of the resulting weights. In many practical tasks sorting is not required, but instead we would like to find out the minimum and the maximum values. Thus, Hamming weight comparators are involved and, composition of different methods described above is helpful.*

Example 3.3. *Suppose we would like to scan and to compare different black and white pictures* $P_1, \ldots, P_X$ *with black and white encoded as '0' and '1' respectively. Thus, any scanned picture is a binary matrix. The Hamming distance* $d(P_i, P_j)$ *between two binary matrices* $P_i \in \{P_1, \ldots, P_X\}$ *and* $P_j \in \{P_1, \ldots, P_X\}$ *indicates the number of mismatched pixels. The distance* $d(P_i, P_j)$ *is the sum of the distances between corresponding lines of* P_i *and* P_j*. The distance with the minimum value indicates two pictures from* $P_1, \ldots, P_X$ *that are the closest match. A similar task needs to be solved to analyze colors in different pictures. In particular, computing the frequency of occurrence of item values permits the dominant color to be identified. As you can see, Hamming distance comparators are involved and the results need to be analyzed by applying operations for discovering the minimum/maximum values and determining how often a given item occurs.*

Example 3.4. *The address-based sort [22] applies to non-repeated integers. The idea is to allocate memory with* 2^M *zero filled one-bit words, where* M *is the size of the data. A new item* I *is recorded by writing the value 1 at the address* I*. As soon as all data items are processed they are sorted and the next task is to extract the sorted data from the memory. Computation of the Hamming weight for the binary vector* $BV(I_b, I_e)$ *beginning*

from the address I_b and ending with the address I_e gives the number of integers in the interval $\{I_b, \ldots, I_e\}$. Further, if the Hamming weight of the vector $BV(I_b, I_e)$ is known, it gives the number of sorted data that have to be extracted which consequently permits the required hardware circuits to be simplified. Many additional problems discussed in [22] can also be solved. If the size of the vector $BV(I_b, I_e)$ is large, it can be split into a set of vectors and presented in the form of a binary matrix that is further processed using the proposed methods.

4. **Hamming Weight Counters and Comparators Based on Look-Up Tables.** A look-up table LUT(n, m) can be used to rapidly implement arbitrary Boolean functions $f_0, \ldots, f_{m-1}$ of n variables $x_0, \ldots, x_{n-1}$. In recent FPGAs (e.g., the 7 series from Xilinx and the Stratix V family from Altera), most often n is 6 and m is either 1 or 2. If we consider the FPGA generations during the last decade, we can see that these values (n, in particular) have been periodically increased. Clearly, h elements LUT(n, m) can be configured to calculate the Hamming weight $w(A)$ of $A = \{a_0, \ldots, a_{N-1}\}$, where $h = \lceil(\log_2(n + 1))/m\rceil$. The idea is to build a network from LUTs(n, m) that can find the Hamming weight for an arbitrary vector A of size N and then to compare this weight with either a fixed threshold κ, or with the Hamming weight $w(B)$ of another binary vector B to be found similarly.

An analysis of practical applications (some of which were described above) shows that the majority require Hamming weight count/comparison for values of N that are divisible by either 8, 32 or 36. Initially we suggest two optimized LUT-based designs that permit the Hamming weight to be found for $N = 8$ (Figure 3(a)) and $N = 36$ (Figure 3(b)). For $N = 32$ either four bits in Figure 3(b) can be assigned to 0 or the results of Figure 3(a) can be incrementally added in a tree-based structure that is composed of the design in Figure 3(a) together with adders, as shown in Figure 4. The Hamming weight for $N > 36$ can be found in a similar tree-based structure (see Figure 4). We also suggest another way to design LUT-based Hamming weight counters/comparators (HWCCs) for $N > 36$. All the designs we are proposing will be evaluated and compared with the existing alternatives both theoretically, and through FPGA-based prototyping.

There are two layers in Figure 3(a) with LUTs$(6, 3)$ and LUTs$(5, 4)$. The first layer counts $W(a_0^i, \ldots, a_5^i)$ and the second layer takes the results of the first layer and determines the 4-bit weight $W(a_0^i, \ldots, a_7^i)$. The delay from the inputs to the outputs (let us designate it Δ_{layers}) is equal to just 2 LUT delays, which is equivalent to 2 gate delays (i.e., $\Delta_{layers} = 2$). The $\surd$ symbols near the layers of LUTs in Figure 3 designate places where such delays arise. There are also two layers in Figure 3(b) with LUTs$(6, 3)$ and two combinational adders. The first layer is composed of 6 LUTs$(6, 3)$ and outputs 6 Hamming weights $W_1, \ldots, W_6$ for 6 sub-vectors $A_1, \ldots, A_6$ of the input vector. The second layer contains 3 LUTs$(6, 3)$ and outputs Hamming weights $\alpha_1\alpha_2\alpha_3$, $\beta_1\beta_2\beta_3$, $\chi_1\chi_2\chi_3$ of the most significant bits (MSB) in $W_1, \ldots, W_6$ ($\alpha_1\alpha_2\alpha_3$), the middle bits in $W_1, \ldots, W_6$ ($\beta_1\beta_2\beta_3$) and the least significant bits (LSB) in $W_1, \ldots, W_6$ ($\chi_1\chi_2\chi_3$). The final result is computed by two combinational adders, as shown in Figure 3(b). Why are conversions similar to the layers 1, 2 not continued after layer 2? We found that any layer with an index greater than $\lceil\log_n N\rceil$ is not cost-effective because either the size of output weights will be increased compared with the previous layers, or the LUTs will not be used efficiently. For example, the Hamming weights can be counted for sub-vectors $\alpha_1\beta_1\chi_1$, $\alpha_2\beta_2\chi_2$, $\alpha_3\beta_3\chi_3$ much like before. In this case we need three additional LUTs$(3, 2)$ and the same LUTs are required for a 1-bit full-adder (FA) [1]. Because carry signals are optimized for arithmetic circuits in FPGAs, from this step on FAs become better than LUT-based converters.

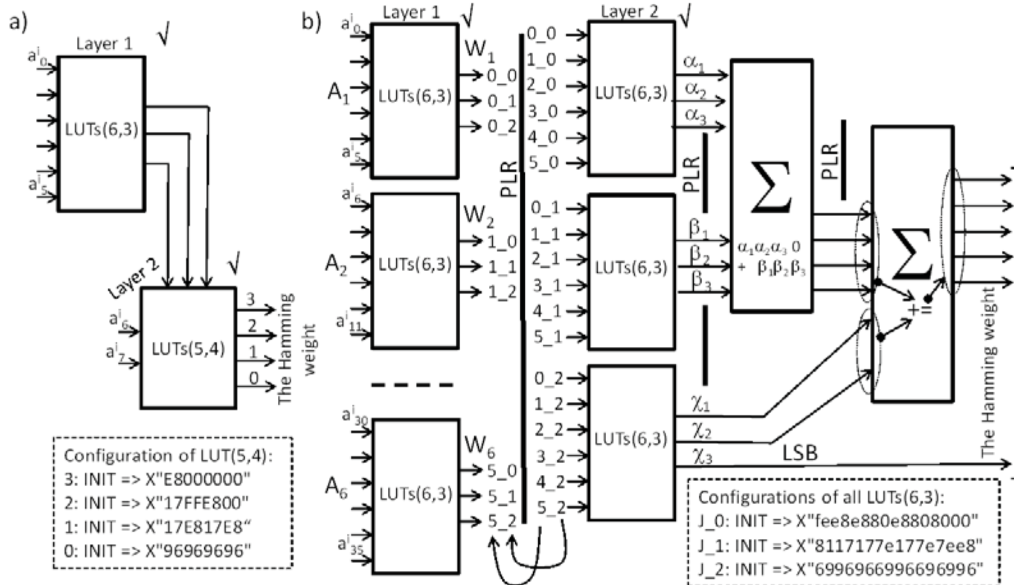

FIGURE 3. Hamming weight counters for $N = 8$ (a) and $N = 36$ (b)

Another possibility is to count the weights of signals from different groups, such as $\alpha_1\beta_1\chi_1\alpha_2\beta_2\chi_2$. This can be done, but the number of LUT outputs is increased compared with the previous layers. The results of analysis and experiments demonstrate that the architecture combining LUTs and adders (see Figure 3(b)) exhibits better cost and latency. Note that all the LUTs in Figure 3(b) are configured identically (see the *INIT* statements defined for the Xilinx ISE in Figure 3(b) for 3 outputs: *J_0* – the top outputs of LUT blocks; *J_1* – the middle outputs; *J_2* – the bottom outputs). The circuit in Figure 3(b) without the output block contains $\lceil(\log_2(n+1))/m\rceil \times (\lceil N/n\rceil + \lceil(N/2)/n\rceil)$ LUTs(n, m). Even for $m = 1$ (the worst case) we need only 27 LUTs. This is negligible, because, for example, the chosen microchip *Zynq xc7z020* of Xilinx contains 13 300 slices and each slice includes 4 LUTs$(6, 1)$. We found that the adders in Figure 3(b) can be implemented in two LUTs$(5, 4)$ with $\Delta_o = 2$ gate level delays. Thus, the total delay Δ for a Hamming weight counter with $N = 36$ is $\Delta = \Delta_{layers} + \Delta_o = 4$ LUT (gate) delays. For the device [37] LUT$(5, 4)$ is built from 2 physically available LUTs$(5, 2)$ (with the total minimum delay 1.382 ns). The available library adders occupy a bit more resources (7 physical LUTs with the total minimum delay 1.795 ns). For the design from [1], the number of FAs is $\nu_{FA} = N - \log_2 N - 1 > 29$ (see Table 1) and the minimal delay is $\log_2 N - 1 >5$ where each unit (from the indicated 5 units) is an FA delay plus the carry signal propagation (this is greater than a LUT (gate) delay). Thus, the proposed design is clearly faster. In further experiments we will show that it is also more economical. The HWCC from [35] has the delay $\log_2 N \times (\log_2 N + 1)/2 > 15$ (see Table 1) and requires significantly more hardware resources.

The comparison of Hamming weights can be done by the carry network (CN) from [1] (see CN for 7 bits in Figure 4(b)) which adds a weight to the 2's-complement of κ (representing '-κ'). Thus, the result of the comparison, c, is 1, if and only if κ is less than $w(A)$. The value κ can be taken as a fixed threshold or as a weight $w(B)$ of another vector B.

If $N > 36$ then HWCCs can be built from several blocks, as shown in Figure 4(a). For example, if $N = 50$ then the design includes one block from Figure 3(b) and two blocks from Figure 3(a), and two unused binary inputs are assigned to 0. If $N = n^L$ $(L = 3, 4, \dots)$

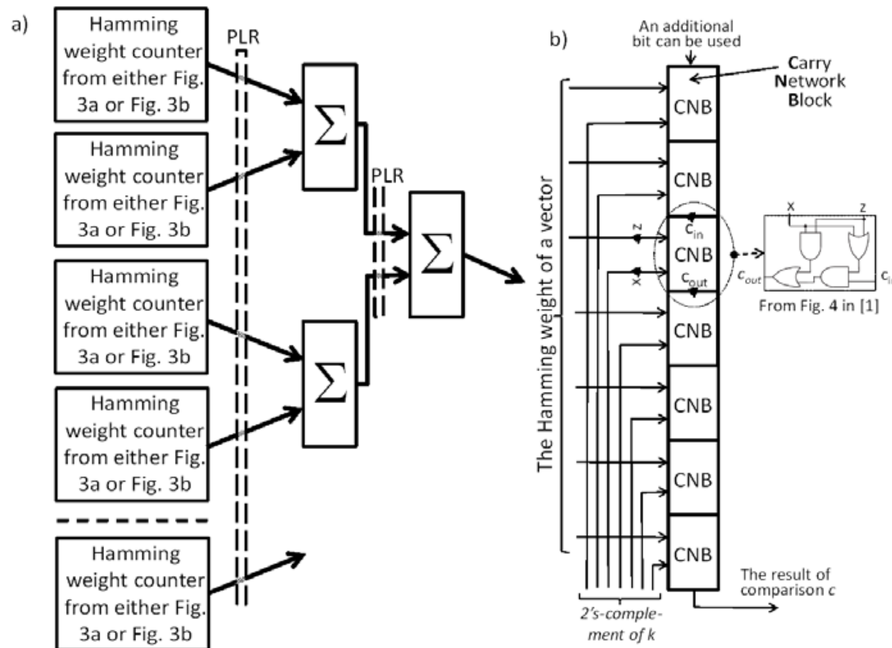

FIGURE 4. Design of Hamming weight counters from LUT-based circuits and adders (a), the carry network (CN) from [1] that performs the comparison (b)

an HWCC can be built similarly to Figure 3(b) for $N = n^2$, i.e., the initially given N-bit vector A is decomposed into $\psi_1 = N/n$ sub-vectors of n-bits. For each sub-vector the Hamming weight is found using LUTs that are organized at the *first layer*. Similar decomposition is applied to $N/2$ outputs of *layer 1* and then to $N/4$ outputs of *layer 2*. Finally the outputs of the *last layer* are added, much like in Figure 3(b). Thus, if $n = 6$, $L = 3$ we build an HWCC for $N = 216$ and then a more complicated HWCC can be constructed from the pre-designed blocks with $N = 8$, $N = 36$ and $N = 216$ using a tree of adders as shown in Figure 4(a).

Let us compare now the proposed HWCC with the existing alternatives [1,9,27,28,35,36]. If $N = n^L$ then the number $C(N) = C(n^L)$ of LUTs in our design is $C(N)\big(\sum_{j=1}^{L}\lceil(\psi_j \times \varphi)$ $/m\rceil\big) + \psi_a$ where $L = \log_n N$, $\psi_1 = N/n$ and for j >1, $\psi_j = (\psi_{j-1} \times \varphi)/n$, where $\varphi = \lceil\log_2(n+1)\rceil$ is the size of each weight on the outputs of any LUT block (we assume that all such blocks are identical), ψ_a is the number of LUTs for the final block of adders. The depth $D(N) = D(n^L)$ of the circuit is $L + D_a$, where D_a is the depth of the final block of adders.

To find the cost and delays for the designs from [1] we need to know the coefficients γ_{FA}, δ_{sum}, δ_{carry} for the expressions in Table 1. In [1] it was assumed that $\gamma_{FA} = 9$, $\delta_{sum} = \delta_{carry} = 2$. If we take these values (9 and 2) to count the LUTs used in FPGA it is undoubtedly not correct (for example, clearly, $\gamma_{FA} <<9$). Thus, we decided to obtain all such values experimentally later on. Initially, and just for the theoretical comparison, the values $\gamma_{FA} = 1$, $\delta_{sum} = \delta_{carry} = 1$ are taken and clearly they represent the best case for [1]. We found that the actual values for the designs from [1] are worse. For the evaluation of the design [35], we used expressions for sorting networks from [17,29,31], which give a better gate count for [35] and are more precise than the expressions from [1]. Values ψ_a for our design are exact because they were taken from the results of synthesis (from Xilinx ISE 14.4). Values D_a are also taken from the synthesis tools. Figure 5 shows the cost/latency comparison in a graphical format.

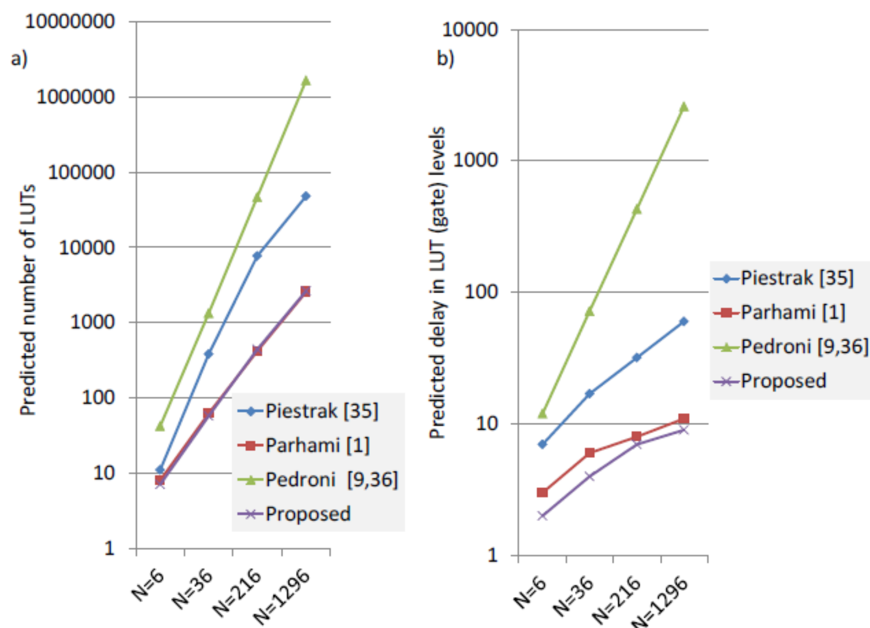

FIGURE 5. Cost (a) and latency (b) comparison for the existing and the proposed designs

An analysis of Figure 5 shows that the proposed designs are better for all values of N. The methods [9,36] are the worst (a similar conclusion is reported in [1]). One advantage of our proposed designs and also the designs from [35] is the ease of pipelining (eventual pipelines for sorting networks were implemented and evaluated in [31]). Pipeline registers (PLR) can be accommodated in Figure 3(b) and Figure 4 in the places indicated by PLR.

5. Hamming Weight Counters and Comparators Based on Counting Networks.

Unlike the existing *sorting networks* (e.g., [17,29-31]), our proposed *counting networks* do not contain comparators. Instead, the basic components are the circuits shown in Figure 6 (i.e., either a *half-adder* or an *XOR* gate). Figure 6 depicts a complete counting network for $N = 64$. The network is composed of 21 segments (each including data independent vertical lines) and 6 levels. The levels calculate the *Hamming weight* of a 2-bit binary vector (level 1), a 4-bit binary vector (level 2), and so on. Thus, the circuit functions like a parallel counter in [1]. However, in contrast to [1], all operations in any segment can be executed in parallel by using simple *AND* and *XOR* gates. In addition, PLRs can be inserted between the segments enabling extremely fast pipelined solutions to be built. The most important characteristic is that the circuit is very regular and easily scalable. Since the counting network computes the Hamming weight, it can be used as a HWCC similarly to the circuits in the previous section.

Let us prove that the networks function properly for any N. Initially we assume that $N = 2^P$ where P is any nonnegative integer: $0, 1, 2, \ldots$ If $N = 1$ the network is obviously correct, since it gives two possible and correct answers: 0 and 1. The maximum value on the outputs of the network for any N is $N = 2^P$. The maximum value on the outputs at any level L ($L = 1, \ldots, P$; $P = \lceil \log_2 N \rceil$) is 2^L. This is the first important characteristic of these networks. The number of outputs of level L ($L > 1$) is always equal to $L + 1$, where L is the number of outputs from any individual circuit of the previous level. This is the second important characteristic of the networks.

The two important characteristics described above allow us to conclude that for any P and $N = 2^P$ the network is correct. Indeed, if we consider any level L, it takes $2 \times L$

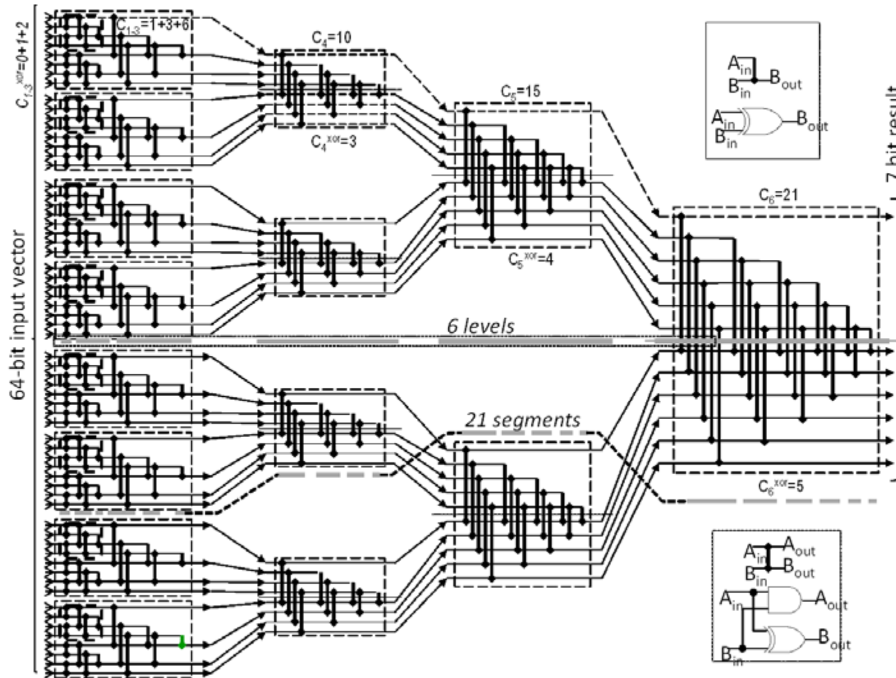

FIGURE 6. Counting network for $N = 64$ where C_L is the number of elements in one block of the level L; C_L^{xor} is the number of XOR gates in one block of the level L.

inputs (two L-bit outputs from the previous level) and computes an $(L + 1)$-bit output. The maximum value on the outputs is 2^L. Thus, the output bits, except for the most significant bit, can be formed at the L bottom lines of the circuit. The most significant bit always has the value 0 except in one case when both inputs are equal to 2^{L-1} and this is obviously correct from the expression $(2^{L-1} + 2^{L-1} = 2^L)$. Thus, the circuit at any level L can be built in such a way that it first adds pairs of values on one-bit lines from two inputs $a_0, \ldots, a_{L-1}$ and $b_0, \ldots, b_{L-1}$ with the same index i, i.e., a_0 and b_0, a_1 and b_1, etc. The value 1 in the upper lines can appear if and only if both bits a_i and b_i are 1. Thus, after the first segment the weight of the line a_0 (in the upper inputs) becomes 2 (input a_0 + input b_0 with weights 1), the weight of the line a_1 becomes 4 (input a_1 + input b_1 with weights 2),..., the weight of the line a_{L-1} becomes 2^L (input a_{L-1} + input b_{L-1} with weights 2^{L-1}). Since the case where both $(a_0, \ldots, a_{L-1})$ and $(b_0, \ldots, b_{L-1})$ are equal to $2^L - 1$ is excluded, the value 1 cannot appear after the segment 1 on the line a_{L-1} and this line is not taken for processing in further segments. In the second segment, all the other lines $a_0, \ldots, a_{L-2}$ are added to the bottom lines $b_0, \ldots, b_{L-1}$ that have the same weights, i.e., the new value a_0 (with the weight 2) is added to b_1 (with the same weight 2), the new value a_1 (with the weight 4) is added to b_2 (with the same weight 4),..., the new value a_{L-2} (with the weight 2^{L-1}) is added to b_{L-1} (with the same weight 2^{L-1}). After the segment 2 the weight of a_0 becomes 4, the weight of a_1 becomes 8, and so on. Similar actions are applied to all the segments, giving the correct sum after the last segment.

The number C_L of the elements in any individual block (i.e., the basic unit of any level) at level L of a combinational counting network is:

$$C_L = \sum_{i=1}^{L} i = L \times \frac{L+1}{2}$$

The total number C of the elements in the entire network with $N = 2^P$ inputs is:

$$C = \sum_{i=1}^{P} C_i \times \frac{N}{2^i}$$

For our example in Figure 6: $C_1 = 1$; $C_2 = 3$; $C_3 = 6$; $C_4 = 10$; $C_5 = 15$; $C_6 = 21$ and $C = 1 \times 32 + 3 \times 16 + 6 \times 8 + 10 \times 4 + 15 \times 2 + 21 \times 1 = 219$.

The delay D_L of any level is: $D_L = L$. The total delay D of the entire network is:

$$D = \sum_{i=1}^{P} i = p \times (p+1)/2$$

For our example in Figure 6: $D_1 = 1$; $D_2 = 2$; $D_3 = 3$; $D_4 = 4$; $D_5 = 5$; $D_6 = 6$ and $D = 21$. Thus, the counting networks are as fast as the best sorting networks.

Let us consider now combinational counting networks for any N (i.e., without the condition $N = 2^P$). Clearly, for any P and $2^{P-1} < N < 2^P$, all unused $2^P - N$ inputs can be assigned to 0, which gives a solution, but this solution is redundant. However, if we remove all elements from the network that do not participate in forming the result, the circuit will function as intended and it becomes non-redundant.

The counting networks are very appropriate for devices that process multiple bits in parallel, such as the *DSP48E1* slice for Xilinx FPGAs [41]. Each slice contains a 48-bit adder/subtractor and implements a number of bitwise logic functions (including *AND* and *XOR*) over 48-bit operands. Since two 48-bit operands can be taken, logic operations over 96-bit vectors are executed non-sequentially in one slice. Taking into account that the networks can be easily pipelined and that a pipeline can be implemented in devices [41] without any additional resources, we expect the proposed counting networks to be very efficient.

6. **Networks with Reusable Cores.** In the introduction we mentioned that one of the objectives of the work is to support run-time operations such as finding minimum/maximum values in the calculated weights, sorting weights, and weight frequency computation. Solving such tasks is required for numerous practical applications discussed in Section 3. Figure 7 demonstrates how these operations can be applied. An input stream/matrix contains binary vectors that need to be analyzed. The vectors can be digitized signals (for example, for the rank-order filter algorithm and median detection [9]), data for combinatorial search algorithms (e.g., matrices representing sets and graphs [14-16]), etc. We would like to analyze such streams/matrices by applying the operations indicated in the introduction and in Figure 7, to the results of Hamming weight/distance counters/comparators that we described in the two previous sections. The distances are measured relative to a given vector (*base for distance* in Figure 7). Comparison is done with a supplied *threshold*. Sorting the input streams enables us to verify the distribution of input vectors relative to the given threshold, to find potential problems in algorithms, to recognize errors (e.g., a wrong threshold), and so on.

High throughput is an important feature for operations in Figure 7 and sorting is considered to be a core operation. Indeed, the result of sorting contains the maximum and minimum values. We will show later that the most frequently occurring item can be found easier in the sorted set. The fastest known parallel sorting methods are based on *even-odd merge* and *bitonic merge* networks [17,29-32]. However, they require enormous hardware resources [4]. The main idea of our proposed method is to find a good compromise between pure combinational parallel operations and sequential stages which execute the parallel operations. The emphasis is on circuits that are as regular as possible and thus easily

scalable, avoiding the need to redirect data by multiplexing operations. An analysis and comparison of different networks shows that *even-odd transition* [4,34] and *max-min* are among the most regular solutions. Thus, we discuss our results based on these. Figure 8 presents examples of such networks (*even-odd transition* in Figure 8(a) and *max-min* in Figure 8(b)) for $N = 8$ data items, which can easily be scaled for any number N. Given input data $(72, 64, 26, 95, 18, 59, 33, 24)$ are sorted in descending order in Figure 8(a). The network in Figure 8(b) finds the items with the maximum and minimum values.

The network in Figure 8(a) contains N vertical lines (levels) of comparators (they are numbered at the top) and each comparator exchanges input values in such a way that the maximum value is placed on the upper line and the minimum value is placed on the bottom line. If data items are swapped they are shown in *italic* and underlined in Figure 8. If there is no exchange at any vertical line then after this line all data are sorted. Hence, the decision about the result can be taken earlier than after propagation through all N lines. The network in Figure 8(b) executes a hierarchical search for the maximum and minimum values. The first vertical level (1) of comparators finds the maximum/minimum values in each pair of input lines. The second level (2) executes similar operations applied to pairs of the maximum/minimum lines from level (1). The iterations are repeated until the overall maximum and the minimum values are found. The number of iterations is $\lceil \log_2 N \rceil$.

The circuits in Figure 8 can be implemented either *non-sequentially* or *sequentially*. Non-sequential (combinational) implementations have many limitations. For example, the results of [31] show that even in the relatively advanced and expensive FPGA XC5VFX130T from the Xilinx Virtex-5 family, the maximum number of input data items (of size $M = 32$ bits) is 64. We suggest an alternative solution to the networks in Figure 8, which is outlined in Figure 9.

The idea is to involve a *feedback register R* and to reuse the same levels sequentially [4], but still applying many parallel operations in the reusable levels. The circuits in Figure 9 have a number of advantages. Hardware resources are obviously decreased. Indeed, the

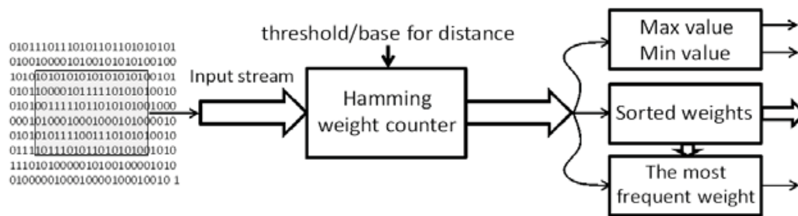

Figure 7. The operations allowing binary streams/matrices to be analyzed

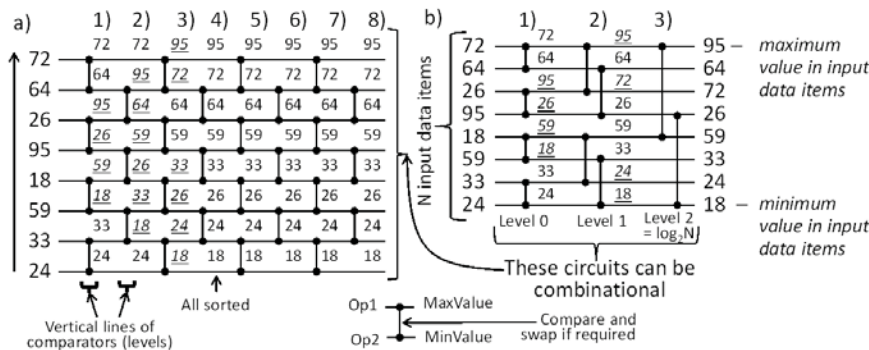

Figure 8. Even-odd transition (a) and max-min (b) networks

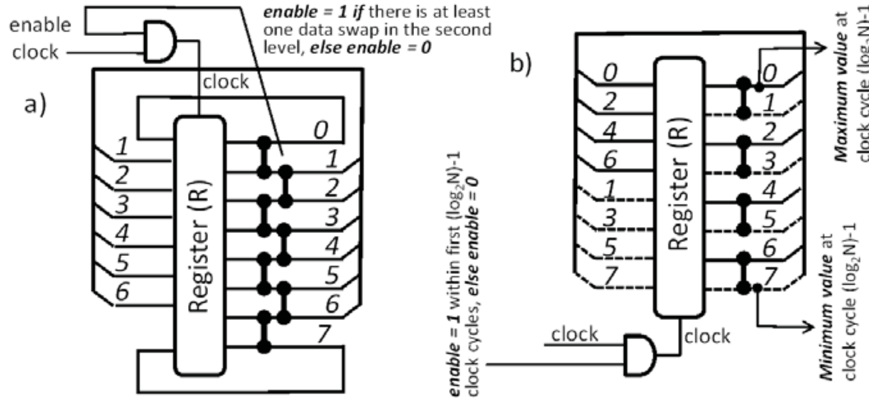

FIGURE 9. Sequential circuits for sorting N data items (a) and finding the minimum and maximum values (b)

networks in Figure 8(a) and in Figure 8(b) require $N \times (N-1)/2$ and $N + \sum_{n=1}^{(\log_2 N)-2} 2^n$ comparators, respectively, whereas the networks in Figure 9(a) and in Figure 9(b) require $N - 1$ and $N/2$ comparators. The implementations of the networks in Figure 9 are very regular, easily scalable for any N and do not involve complex interconnections. The number of paths through the vertical levels of comparators is decreased. Indeed, the result of sorting in Figure 8(a) is produced at level 4, but since the network is hardwired, the remaining levels *5-8 are involved*, causing unnecessary *4 paths* and *4 additional propagation delays*. The network in Figure 9(a) does not involve additional iterations. As soon as the *enable* signal that is produced at the odd (the rightmost) level is 0, sorting is finished [4]. Since the depth of comparators is just 2 in Figure 9(a) and just 1 in Figure 9(b), the propagation delay is negligible.

The network in Figure 9(a) sorts N input data items in T_s clock cycles and $T_s \leq N/2$ [34]. Indeed, there are $N/2$ even-odd levels in Figure 8(a) [34] and the number of cycles in Figure 9(a) is less than or equal to $N/2$ because the result can be produced before passing sequentially through all the levels (such as that depicted in Figure 8(a)). The network in Figure 9(b) finds the minimum and maximum values in T_f clock cycles and $T_f = (\lceil \log_2 N \rceil) - 1$. Indeed, at the iteration $(\lceil \log_2 N \rceil) - 1$ the results are ready on the outputs of the combinational comparators. In the next section we will compare the proposed networks with the known alternatives.

Suppose we have a set of N sorted weights which eventually include repeated items and we need the most frequently repeated item to be found. The proposed solution for this problem is shown in Figure 10 where $N - 1$ comparators form a binary vector. The most frequently repeated item can be discovered if we find the maximum number of consecutive ones in the vector and take the item from any input of the comparators that forms the sub-vector with the maximum number of successive ones.

The binary vector that represents the result of comparison is saved in the *feedback register R*. The right-hand circuit in Figure 10 implements the method described above which enables the same combinational unit (such as that composed of *AND* gates in Figure 10) to be reused iteratively in each subsequent clock cycle. This forces any intermediate binary vector that is formed on the outputs of the *AND* gates to be stored in the *register R*. Hence, any new clock cycle reduces the maximum number of consecutive ones $O_{\max}$ in the vector by one and as soon as all outputs of the *AND* gates are set to 0 we can conclude that $O_{\max} = \xi + 1$, where ξ is the number of the last clock cycle. Indeed, when there is just one value 1 in the *register*, all the outputs of the *AND* gates are set to 0 and

an additional clock cycle is not required to reach a conclusion. The index of the single 1 in the *register* is the index (position) of the first value 1 (from the top) in the set with $O_{\max}$. The feedback from the outputs of the *AND* gates enables any intermediate binary vector to be stored in the *register*. Not all the gates are entirely reused. At the first step there are $N-1$ active gates. In each subsequent clock the number of such gates is decremented because the lower gate is blocked by 0 to be written to the bottom bit of the *register*. In each new clock cycle, this zero always propagates to an upper position and blocks another gate. Much like the circuit in Figure 9, the circuit in Figure 10 is very simple and fast. It is composed of just $N-1$ *AND* gates, the *register* R, and minimal supplementary logic. Thus the maximum attainable clock frequency is high, as we will show in the next section. Additionally, combining combinational (a set of *AND* gates) and sequential techniques enables the results to be obtained very fast. Certainly, there is no propagation through redundant paths. The result is ready when all outputs of the *AND* gates are zeros.

7. **Experiments and Comparisons.** We present in this section a thorough evaluation and comparison of the proposed designs. The charts in Figure 11 compare the architectures suggested with the known alternatives from [1,9,27,28,35,36]. All the circuits were synthesized, implemented, and tested in the Xilinx *Zynq xc7z020* microchip. The first chart (Figure 11(a)) shows the maximum combinational path delay in pure combinational implementations of different Hamming weight counters. The second chart (Figure 11(b)) indicates the number of FPGA slices for different designs. The total number of available slices in the microchip *xc7z020* is 13 300. For our circuits we considered pipelined implementations which include additional *PLRs* between layers of LUT-based designs and between some levels of counting networks. We found that the maximum delay between *PLRs* can be as little as 1.3ns for the LUT-based designs and 0.7ns in counting networks. Thus, potential throughput can be less than 2ns per weight in LUT-based designs and less than 1ns per weight in counting networks. The latter might be the best solution for pipelined implementations where the registers are inserted between all vertical lines (see Figure 6). Even without pipelining, the proposed LUT-based designs are the best in

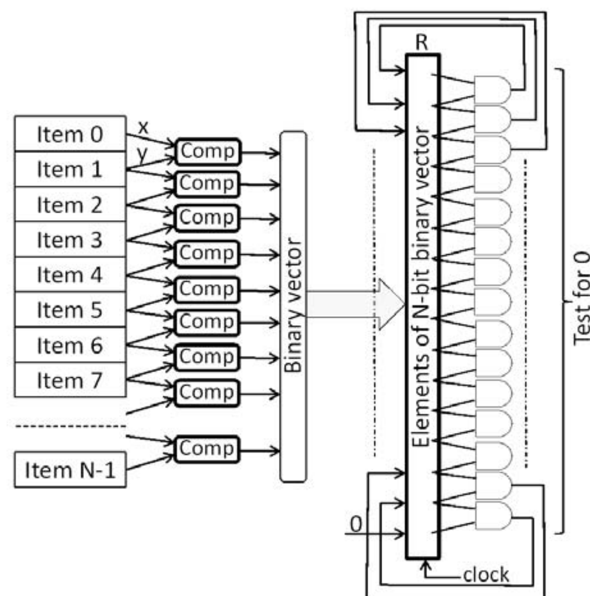

FIGURE 10. Most frequent weight computation in a given sorted set of weights

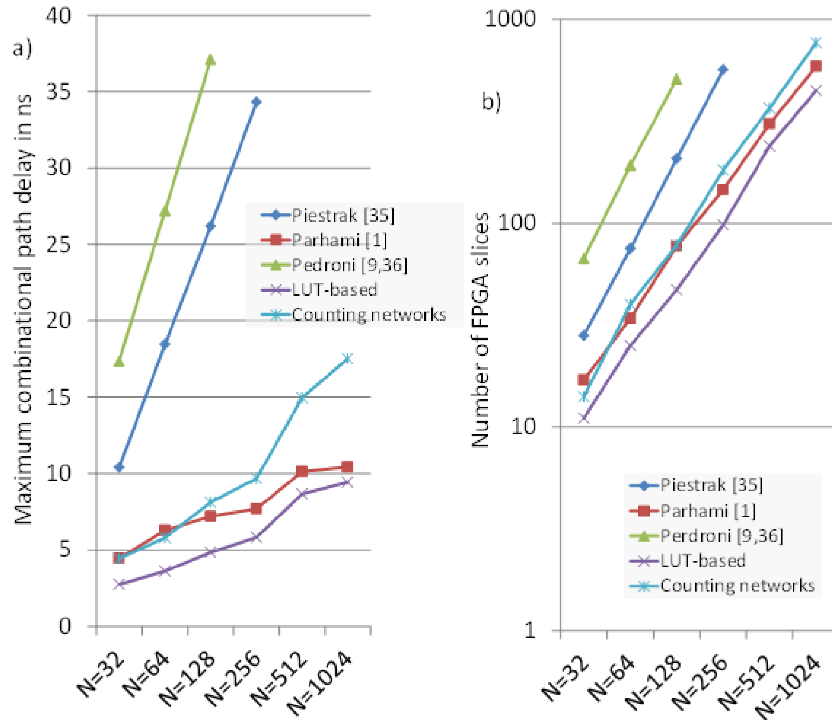

FIGURE 11. Latency (a) and cost (b) comparison from the result of experiments

both latency (see Figure 11(a)) and cost (see Figure 11(b)). Some inconsistencies with theoretical results [1,9,27,28,35,36] have appeared because routing involves additional resources and delays which are not the same for different designs and these are not taken into consideration in theoretical expressions [1,17,31]. The worst routing results are for the designs [9,35,36]. We think that this is due to the irregularity of the network (details are given in [4]). In particular, the actual delay for the design [35] is significantly worse than the predicted delay for the best sorting networks [17,29,31]. Finally, we decided that it made no sense to continue experiments for the designs [9,35,36] when $N \geq 512$. This conclusion is in conformity with the results [1]. The relationship between LUT-based circuits and the designs from [1] is more or less what we expected, although the delays for [1] are, indeed, not bad. We believe that this is because of the availability of highly optimized arithmetic-targeted circuits in FPGAs.

We compared the proposed reusable circuits shown in Figure 9 with the *even-odd merge* networks, which have the same latency as *bitonic merge* networks (and a little bit lower cost). The size M of data items was set to 16. The value $M = 16$ permits weights of up to $N = 2^{16}$ to be compared, which exceeds the requirements of any practical problem. Figure 12 presents the comparison charts. Note that the circuit in Figure 9 forms the result through the sequential activation of two levels (i.e., an *odd* and an *even* level) and all the comparators of these levels are active in parallel. Thus, the output is ready after only T_s clock cycles. The maximum value of T_s ($T_{\max}$) in the worst case is equal to $N/2$ [34]. This value (i.e., T_s) is almost always smaller because the result is ready as soon as the *enable* signal (see Figure 9(a)) is equal to 0. We have not taken into account the eventual reduction in T_s but have selected $T_s = N/2$ (i.e., the worst case for our designs) as a coefficient so as to normalize the results of comparison (see the chart marked with "$Proposed \times T_s$" in Figure 12(a)).

From Figure 12 we can see that up to $N = 40$ the proposed design is the best in both cost and latency. If $N > 40$ the latency is better for the *even-odd merge* network.

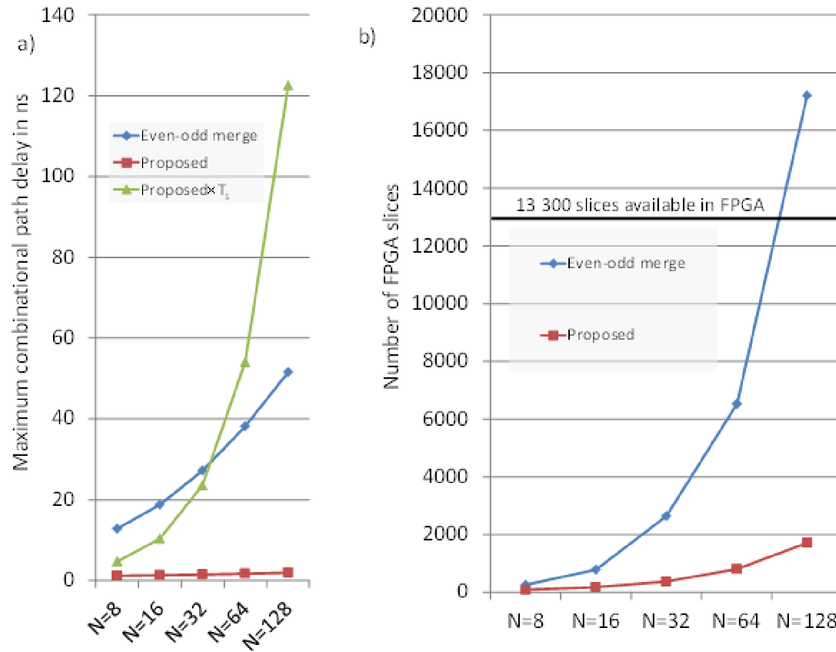

Figure 12. Comparison of the even-odd merge and the proposed networks

However, the main problem is the cost (i.e., the resources consumed). While our design for $N = 40$ requires about 3.5% of FPGA slices, the *even-odd merge* consumes 27% and for $N = 128$ it cannot be implemented at all due to the lack of FPGA resources (see the horizontal line indicating the number 13 300 of available FPGA slices in Figure 12(b)). For our reusable network in Figure 9(a) we were able to synthesize, implement, and test the circuit for $N = 512$ ($M = 16$), which requires a bit more than 50% of the FPGA resources and the remaining slices are sufficient to implement other circuits that may be needed for an analyzer of binary matrices/streams. If we consider matrices of size 1000×1000 and take into account that M can be reduced, with values for M up to 10 the results of the complete analysis (including the Hamming weight computation and sorting for lines/columns) are ready after less than 0.5 μs and the analyzer can be built on one *Zynq xc7z020* microchip.

Sorted data can be used to find the most frequent weight with the aid of the circuit shown in Figure 10. The required resources are small. We implemented and tested the circuit for $N = 1024$, $M = 16$ and it consumes 412 slices and works with the maximum attainable frequency 274.2MHz.

If only the maximum and/or the minimum values need to be found, then the circuit can be built in the way shown in Figure 9(b). It consumes fewer resources and is faster. We synthesized, implemented, and tested the circuit for $N = 1024$, $M = 16$ and it consumes 4809 slices and operates with the maximum attainable frequency 657.4MHz.

We also implemented and tested a combined DSP-based architecture which includes a counting network and a set of adders. The circuit contains 11 embedded DSP slices DSP48E1 [41] (from 220 available slices in the used *Zynq xc7z020* microchip). The first 4 segments of the network (see Figure 6) were implemented in 7 DSP slices. Two segments (5 and 6) were built as counting networks on FPGA logical slices. The remaining part employs 4 DSP slices configured as 15 independent 12-bit adders. The total number of logical slices in the entire circuit is only 37. Thus, the cost is small. Since DSP slices [41] implement pipelining without the need for extra resources, the latency is also very good.

From the previous results we can conclude that the proposed solutions are better than the best known alternatives and they permit digital Hamming weight and distance analyzers for binary vectors and matrices to be implemented in FPGA devices.

8. **Architecture and Practical Implementation of the Analyzer.** It is practical to combine software and hardware operations. For example, a processing system (PS) might collect data from different sources (i.e., receive the results of measurements from sensors shown in Figure 1) and execute such operations over the received data that do not involve parallel computations. As soon as it is necessary to perform fast parallel operations (such as those shown in Figures 3, 4, 6, 8-10) the relevant data can be transferred to programmable logic (PL) which executes parallel processing in hardware. The results of the parallel processing are transferred back to the PS. Combining the capabilities of software and hardware permits many characteristics of developed applications to be improved. The earliest work in this direction was done at the University of California at Los Angeles [42]. The idea was to create *Fixed + Variable* structure computer and to augment a standard processor by an array of reconfigurable logic, assuming that this logic can be utilized to solve some processor tasks faster and more efficiently. Such a combination of the flexibility of software and the speed of hardware was considered to be a new way to evolve higher performance computing from any general purpose computer. The level of technology in 1959-1960 did not permit this idea to be put in practice. Today a very similar technique was implemented on a chip combining multi-core processors, embedded blocks, and advanced reconfigurable logic. Figure 13 presents the main components of the Xilinx Zynq xc7z020 Extensible Processing Platform (EPP) [37,43].

PS executes software programs that can be developed in the C/C++ languages. PL is an Artix-7 family [43] FPGA implemented on the same microchip as PS. PS and PL can exchange data using AXI (Advanced eXtensible Interface)-based high-bandwidth connectivity. We used the xc7z020 microchip that is available on the ZedBoard [44] for tests and experiments. Thus, in the same microchip we implemented and tested:

- Systems requiring the development of software and invoking on-chip processing blocks.
- Application-specific hardware in programmable logic using embedded blocks (such as DSP slices and memories) and arbitrary logic composed of slices and flip-flops.
- A *Fixed+Variable* structure computational system that combines a PS and a PL with high-speed data exchange between them through the AXI-based interface.

Figure 14 describes the proposed scenarios of interactions between the PS and the PL. Two types of computations were implemented and verified. In the first type (see Figure 14(a)) the PL functions as an autonomous system receiving data streams in the form of vectors or matrices from external pins, executing the operations described in the previous sections over the data, and transferring the results either to external pins or to the PS. In the second type (see Figure 14(b)) the PL is considered to be a slave sub-system of the PS. As soon as the PS needs to accelerate operations, it sends a request to the PL and transfers data associated with the operations to the PL. The PL executes the operations and informs the PS as soon as the results are ready. Finally, the results are sent back to the PS.

Figure 15 presents the architecture of the final analyzer which includes: the accelerators proposed and completed within this work; the interfacing circuit; and a sub-system implemented in software of the PS (see Figure 13 and the block in Figure 15 surrounded by a dashed line). The accelerators are the devices described in this paper:

- The Hamming weights counter (see Sections 4 and 5).

- The Hamming weight comparator comparing either the weight of a vector A with a given threshold κ or the weights of two vectors A and B (see Section 4).
- A Hamming distance counter which is based on the Hamming weights counter and *XOR* gates (see Sections 1 and 4).
- A sorter, outputting the sorted weights from a given binary matrix/stream (see Section 6).
- A searcher for the maximum and/or the minimum values (see Section 6).
- A searcher for the most frequently occurring item (weight) in the sorted sequence (see Section 6).

All the accelerators were described in VHDL using *generate* and *generic* statements. The code is easily parameterized for different values of N and M and, thus, can be used for numerous practical applications referenced in Sections 1-3.

Figure 16 gives more details of the interaction between the PS and the PL which is organized with the aid of Xillybus Lite IP core [45]. The user software applications run in the ARM Cortex-A9 under Linux. The accelerators are designed in Xilinx ISE 14.4 and they interact with the Xillybus IP core as shown in Figure 16. The latter provides data exchange with the PS through AXI.

Software applications were developed in C and they execute the following tasks: 1) getting data from the host PC; 2) partitioning the data and transmitting them to the PL

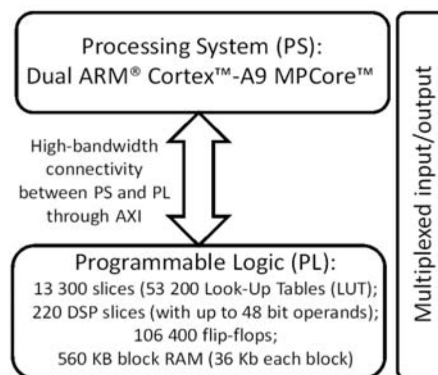

FIGURE 13. The main components of xc7z020 extensible processing platform

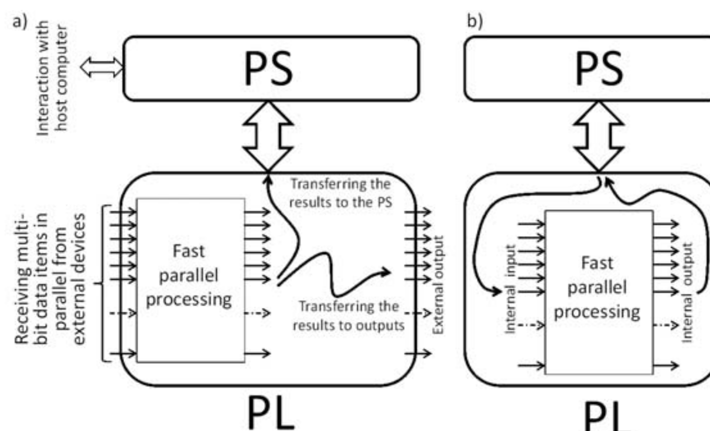

FIGURE 14. Interactions between PS and PL: PL provides fast parallel processing of external data and PS can use the results (a); PL executes dedicated operations for PS on internal requests from PS (b).

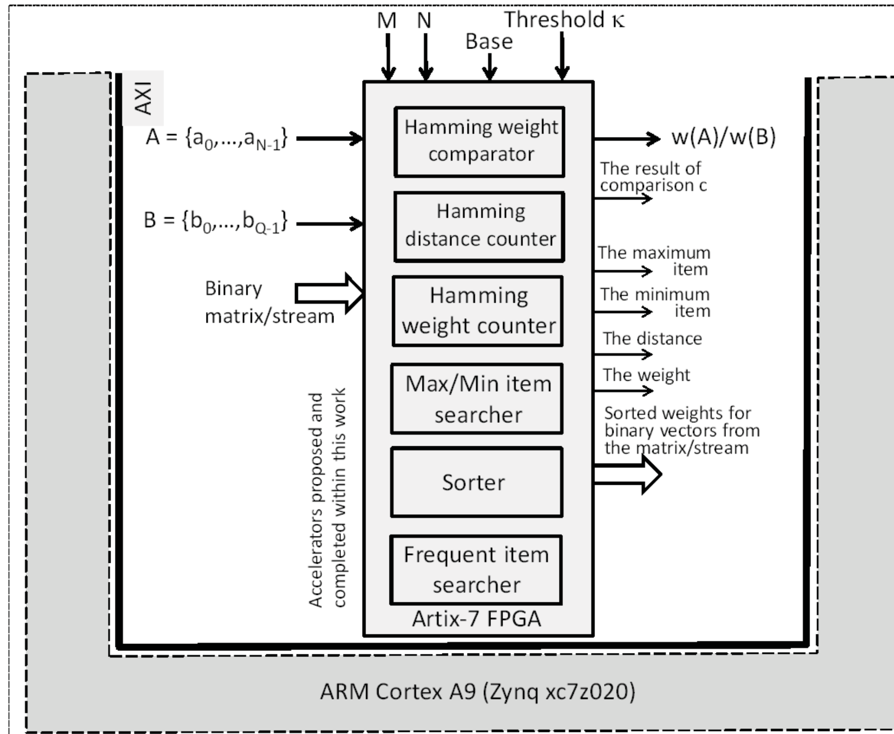

FIGURE 15. Architecture of the proposed analyzer

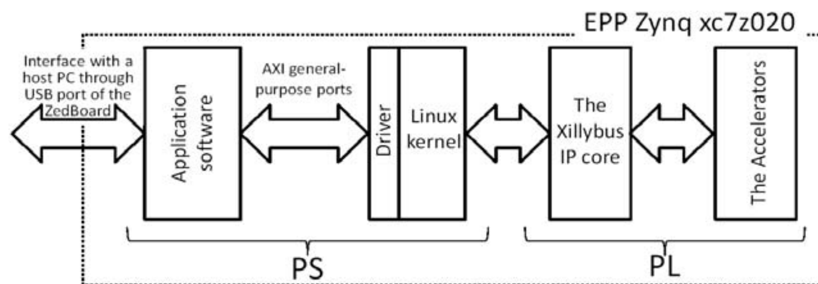

FIGURE 16. Implemented interaction between PS and PL

when required; 3) getting an application-specific analysis of the results from the PL (see Section 3); 4) support for experiments with the developed hardware in the PL. Accelerator projects in the PL can be configured to support both the types of computations shown in Figure 14, i.e., initial stream is uploaded either from external pins or internally from the PS through AXI. Since the PL hardware can be modified and improved, the proposed digital Hamming weight and distance analyzers for binary vectors and matrices can be seen as an example of an on-chip *Fixed+Variable* structure computational system.

Let us consider an example that demonstrates the efficiency of the analyzer. Suppose we want to find a minimal row cover of a given binary matrix, i.e., a minimum number of rows such that in conjunction they have at least one value '1' in each column. The approximate algorithm [14] that allows this problem to be solved requires the following sequence of steps (see Figure 17(a)): 1) discovering a matrix column $C_{\min}$, with the minimal Hamming weight $N^1_{\min}$ (if $N^1_{\min} = 0$ then the covering does not exist); 2) discovering a row $R_{\max}$, with the value '1' in the column $C_{\min}$, with the maximum Hamming weight $N^1_{\max}$; 3) removing the row $R_{\max}$ and all the columns, which have values '1' in the $R_{\max}$; 4) repeating the

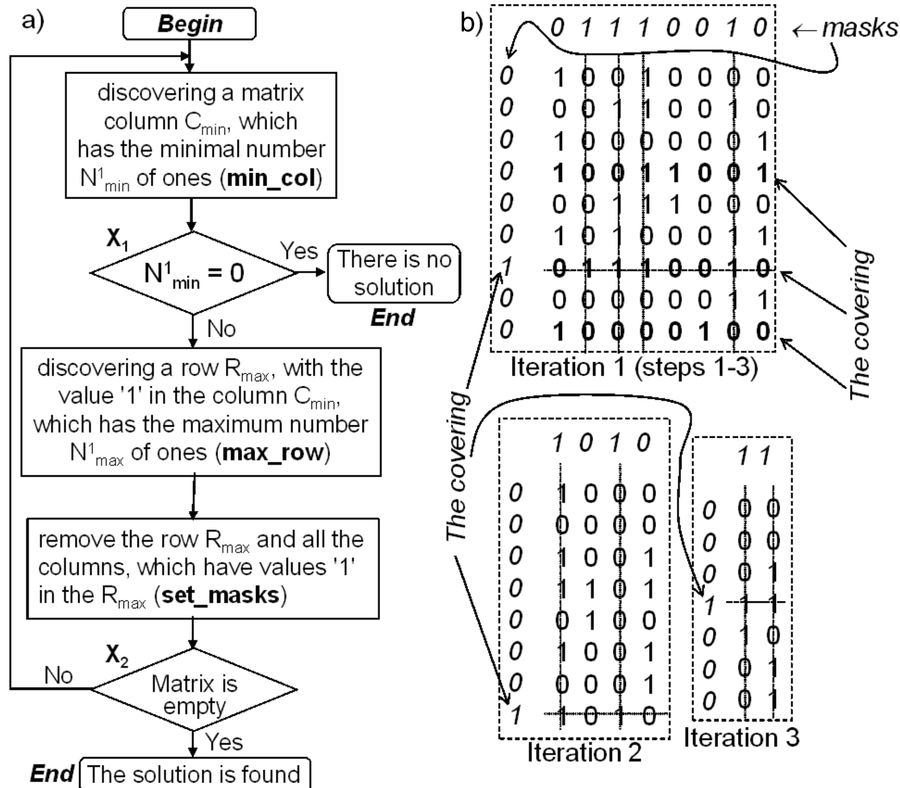

FIGURE 17. Approximate matrix covering algorithm (a) and iterations of the algorithm applied to the given binary matrix (b)

steps 1-3 until the matrix is empty. Figure 17(b) gives an example of a particular matrix to which the steps in Figure 17(a) have been applied.

The proposed analyzer (see Figures 15 and 16) can be used as follows:

- The PS receives the given matrix from the host PC.
- The matrix is transmitted to the PL and all horizontal and vertical masks (see Figure 17(b)) are reset to zero.
- The PL executes step 1 and sends the values $N^1_{\min}$ and $C_{\min}$ to the PS;
- If $N^1_{\min} = 0$, the PS informs the host PC that there is no solutions, otherwise it indicates for which rows the value $N^1_{\max}$ has to be found in the PL;
- The PL finds $R_{\max}$ and sends it to the PS;
- The PS updates masks in the PL. The masks are used to indicate the rows and columns that have been removed and, thus, the same masked (reduced) matrix is taken for subsequent steps;
- The steps above are repeated until the covering is found or until it is concluded that the solution does not exist.

Experiments with arbitrary generated matrices (in total more than 100) have demonstrated that the result is found faster than in software [14]. Comparing with the implementation of the algorithm in Figure 17(a) just in reconfigurable logic (see, for example, [46]) we found that:

1. The EPP-based implementation is more regular and easier scalable;
2. The number of FPGA components is reduced because a part of the job is done in software;
3. Throughput is increased because of the rational hardware/software co-design and parallel execution of the operations in software and in hardware.

Similar experiments were carried out with some practical applications from Section 3. For all of them the results were better than for alternative implementations that do not involve our proposed methods (see Sections 4-6). The methods themselves were additionally tested and compared in Section 7.

9. **Conclusion.** The paper is dedicated to digital Hamming weight and distance analyzers for binary vectors and matrices and suggests:

1. LUT-based circuits and counting networks for *Hamming weight/distance counters/comparators* that are faster and less resource consuming than the best known alternatives;
2. Sorters/searchers for weights in sets of vectors originating as binary matrices or streams. These combine combinational with sequential operations to provide better solutions for the applications considered than the popular *even-odd merge* and *bitonic merge* networks;
3. An EPP-based implementation of the analyzers that combines software and reconfigurable hardware, which can be seen as an example of an on-chip *Fixed+Variable* structure computational system.

The architecture of the analyzer was implemented and verified in commercially available microchips. The results were compared both theoretically and practically. The practical analysis consisted of numerous experiments using the most recent extensible processing platform combining a processing system and reconfigurable logic. The experiments comprehensively demonstrated that the analyzers incorporating our novel designs outperformed the currently published alternatives by a significant margin. It is also shown that the analyzer and its accelerating modules can be used in numerous practical applications in such areas as digital signal, image and data processing, coding and error correction, cryptography, and combinatorial search.

Acknowledgments. The authors would like to thank Ivor Horton for his very useful comments and suggestions and to Artjom Rjabov for making some experiments in EPP. This research was supported by FEDER through the Operational Program Competitiveness Factors – COMPETE and by National Funds through FCT – Foundation for Science and Technology in the context of project FCOMP-01-0124-FEDER-022682 (FCT reference PEst-C/EEI/UI0127/2011).

REFERENCES

[1] B. Parhami, Efficient Hamming weight comparators for binary vectors based on accumulative and up/down parallel counters, *IEEE Transactions on Circuits and Systems – II: Express Briefs*, vol.56, no.2, pp.167-171, 2009.

[2] P. D. Wendt, E. J. Coyle and N. C. Gallagher, Stack filters, *IEEE Transactions on Acoustics, Speech, Signal Processing*, vol.34, no.4, pp.898-908, 1986.

[3] C. Qin, C. C. Chang and P. L. Tsou, Perceptual image hashing based on the error diffusion halftone mechanism, *International Journal of Innovative Computing, Information and Control*, vol.8, no.9, pp.6161-6172, 2012.

[4] V. Sklyarov and I. Skliarova, Data processing in FPGA-based systems, tutorial, *Proc. of the 6th International Conference on Application of Information and Communication Technologies*, pp.291-295, 2012.

[5] S. Yang, R. W. Yeung and C. K. Ngai, Refined coding bounds and code constructions for coherent network error correction, *IEEE Transactions on Information Theory*, vol.57, no.3, pp.1409-1424, 2011.

[6] C. K. Ngai, R. W. Yeung and Z. Zhang, Network generalized Hamming weight, *IEEE Transactions on Information Theory*, vol.57, no.2, pp.1136-1143, 2011.

[7] I. Skliarova and A. B. Ferrari, Reconfigurable hardware SAT solvers: A survey of systems, *IEEE Transactions on Computers*, vol.53, no.11, pp.1449-1461, 2004.
[8] O. Milenkovic and N. Kashyap, On the design of codes for DNA computing, *Proc. of International Conference on Coding and Cryptography*, pp.100-119, 2005.
[9] V. Pedroni, Compact Hamming-comparator-based rank order filter for digital VLSI and FPGA implementations, *Proc. of the IEEE International Symposium on Circuits and Systems*, pp.585-588, 2004.
[10] K. Chen, Bit-serial realizations of a class of nonlinear filters based on positive Boolean functions, *IEEE Transactions on Circuits and Systems*, vol.36, no.6, pp.785-794, 1989.
[11] M. Storace and T. Poggi, Digital architectures realizing piecewise-linear multivariate functions: Two FPGA implementations, *International Journal of Circuit Theory and Applications*, vol.39, no.1, pp.1-15, 2011.
[12] K. Asada, S. Kumatsu and M. Ikeda, Associative memory with minimum hamming distance detector and its application to bus data encoding, *Proc. of IEEE Asia-Pacific Application-Specific Integrated Circuits Conference*, 1999.
[13] C. Barral, J. S. Coron and D. Naccache, Externalized fingerprint matching, *Proc. of the 1st International Conference on Biometric Authentication*, pp.309-315, 2004.
[14] A. Zakrevskij, Y. Pottoson and L. Cheremisiniva, *Combinatorial Algorithms of Discrete Mathematics*, Tallinn, TUT Press, 2008.
[15] I. Skliarova and A. B. Ferrari, A software/reconfigurable hardware SAT solver, *IEEE Transactions on Very Large Scale Integration Systems*, vol.12, no.4, pp.408-419, 2004.
[16] I. Skliarova and A. B. Ferrari, The design and implementation of a reconfigurable processor for problems of combinatorial computation, *Journal of Systems Architecture, Special Issue on Reconfigurable Systems*, vol.49, no.4-6, pp.211-226, 2003.
[17] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching*, 1973.
[18] V. A. Vaishampayan, *Query Matrices for Retrieving Binary Vectors Based on the Hamming Distance Oracle*, http://arxiv.org/pdf/1202.2794v1.pdf.
[19] R. Ma and S. Cheng, The universality of generalized Hamming code for multiple sources, *IEEE Transactions on Communications*, vol.59, no.10, pp.2641-2647, 2011.
[20] A. X. Liu, K. Shen and E. Torng, Large scale Hamming distance query processing, *Proc. of the 27th International Conference on Data Engineering*, pp.553-564, 2011.
[21] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*, John Wiley and Sons, 2011.
[22] V. Sklyarov, I. Skliarova, D. Mihhailov and A. Sudnitson, Implementation in FPGA of address-based data sorting, *Proc. of the 21st International Conference on Field-Programmable Logic and Applications*, pp.405-410, 2011.
[23] V. Sklyarov and I. Skliarova, Modeling, design, and implementation of a priority buffer for embedded systems, *Proc. of the 7th Asian Control Conference*, pp.9-14, 2009.
[24] J. Gu, P. W. Purdom, J. Franco and B. W. Wah, Algorithms for the satisfiability (SAT) problem: A survey, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol.35, pp.19-151, 1997.
[25] Z. Yin, C. Chang and Y. Zhang, An information hiding scheme based on $(7, 4)$ Hamming code oriented wet paper codes, *International Journal of Innovative Computing, Information and Control*, vol.6, no.7, pp.3121-3130, 2010.
[26] R. D. Lin, T. H. Chen, C. C. Huang, W. B. Lee and W. S. E. Chen, A secure image authentication scheme with tampering proof and remedy based on Hamming code, *International Journal of Innovative Computing, Information and Control*, vol.5, no.9, pp.2603-2617, 2009.
[27] E. E. Swartzlander, Parallel counters, *IEEE Transactions on Computers*, vol.C-22, no.11, pp.1021-1024, 1973.
[28] B. Parhami and C.-H. Yeh, Accumulative parallel counters, *Proc. of Asilomar Conference on Signals, Systems & Computers*, pp.966-970, 1995.
[29] K. E. Batcher, Sorting networks and their applications, *Proc. of AFIPS Spring Joint Computer Conference*, pp.307-314, 1968.
[30] G. Gapannini, F. Silvestri and R. Baraglia, Sorting on GPU for large scale datasets: A thorough comparison, *Information Processing and Management*, vol.48, no.5, pp.903-917, 2012.
[31] R. Mueller, J. Teubner and G. Alonso, Sorting networks on FPGAs, *The International Journal on Very Large Data Bases*, vol.21, no.1, pp.1-23, 2012.
[32] M. Zuluada, P. Milder and M. Puschel, Computer generation of streaming sorting networks, *Proc. of the 49th Design Automation Conference*, pp.1245-1253, 2012.

[33] R. D. Chamberlain and N. Ganesan, Sorting on architecturally diverse computer systems, *Proc. of the 3rd Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pp.39-46, 2009.
[34] *GPU Gems, Improved GPU Sorting*, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.
[35] S. J. Piestrak, Efficient Hamming weight comparators of binary vectors, *Electronic Letters*, vol.43, no.11, pp.611-612, 2007.
[36] V. A. Pedroni, Compact fixed-threshold and two-vector Hamming comparators, *Electronic Letters*, vol.39, no.24, pp.1705-1706, 2003.
[37] Xilinx, *ZedBoard: Zynq-7000 AP SoC Concepts, Tools, and Techniques*, 2012.
[38] L. Field, T. Barnie, J. Blundy, R. A. Brooker, D. Keir, E. Lewi and K. Saunders, Integrated field, satellite and petrological observations of the November 2010 eruption of Erta Ale, *Bulletin of Volcanology*, vol.74, no.10, pp.2251-2271, 2012.
[39] I. Skliarova and V. Sklyarov, Design methods for FPGA-based implementation of combinatorial search algorithms, *Proc. of Int. Workshop on SoC Design*, pp.359-368, 2006.
[40] J. D. Davis, Z. Tan, F. Yu and L. Zhang, A practical reconfigurable hardware accelerator for Boolean satisfiability solvers, *Proc. of the 45th ACM/IEEE*, pp.780-785, 2008.
[41] Xilinx, *7 Series DSP48E1 Slice User Guide*, 2012.
[42] G. Estrin, Organization of computer systems – The fixed plus variable structure computer, *Proc. of Western Joint IRE-AIEE-ACM Computer Conference*, pp.33-40, 1960.
[43] Xilinx, *Zynq-7000 All Programmable SoC First Generation Architecture*, http://www.xilinx.com/support/documentation/data_sheets/ds188-XA-Zynq-7000-Overview.pdf.
[44] ZedBoard, *Zynq™ Evaluation and Development Hardware User's Guide*, https://www.avnet.co.jp/sitecore/shell/Controls/Rich%20Text%20Editor/~/media/555A3D3BFBC74EA4B0488A74388F983C.ashx.
[45] Xillybus, *Xillybus Lite for Zynq-7000: Easy FPGA registers with Linux*, http://xillybus.com/xillybus-lite.
[46] V. Sklyarov, I. Skliarova and B. Pimentel, FPGA-based implementation and comparison of recursive and iterative algorithms, *Proc. of the 15th International Conference on Field-Programmable Logic and Applications*, pp.235-240, 2005.