BIT-PLANE STACK FILTER ALGORITHM FOR FOCAL PLANE PROCESSORS

Andrés Frías-Velázquez, Wilfried Philips

Ghent University-TELIN-IPI-IBBT, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium

ABSTRACT

This work presents a novel parallel technique to implement stack morphological filters for image processing. The method relies on applying the image bitwise decomposition to manipulate the grayscale image at a bit-plane level, while simple logical operations and Positive Boolean Functions (PBF's) are executed in parallel to derive the transformed bit-planes. The relationship between the bitwise and threshold decomposition is closely investigated and analysed, which lead us to derive an algorithm whose control flow is full binary encoded. Furthermore, the algorithm exhibits an interesting performance, which depends on the image histogram thanks to its hierarchical processing and the study of the relationship among binary decompositions.

Index Terms— Stack filters, Morphological operators, Rank filters, Bitwise decomposition, SIMD array.

1. INTRODUCTION

Stack filters are defined as the class of non-linear digital filters that can be implemented by decomposing a multilevel signal into a set of either binary or low dynamic range signals through a monotone increasing mapping and then restoring it by summing up the transformed signals. These two properties introduced in [1] are commonly called threshold decomposition (TD) and stacking, respectively. Weighted order statistic (WOS) filters, rank order filters (ROF), morphological filters can be implemented as stack filters. These sort of filters have been especially exploited on areas such as image and speech restoration by removing impulsive and non-Gaussian noise. Other areas of prolific success are pattern recognition and medical image processing.

Apart from the theoretical interest, the threshold decomposition and stacking property have enabled implementing stack filters in a efficient way. There is plenty of literature dealing with the implementation of stack filters for diverse filter specifications, VLSI architectures, and applications. Most of these approaches are surveyed in [2], covering techniques having a bit-serial processing in mind.

On the other hand, word-parallel implementations such as [3, 4, 5] have attempted to reduce the latency experienced on bit-serial implementations at a expense of a higher chip area.

Parallel architectures such as Focal Plane Processors (FPPs) allow performing low-level image processing in parallel on all image pixels. This new capability turns the traditional bit-parallel processing into bit-plane processing, leading to new techniques to implement basic image processing algorithms such as stack filters.

In this paper, we propose a serial implementation for stack filters and then we extend it to the FPP architecture. As a result, we employ the image bitwise decomposition to deal with bit-planes and take advantage of the very fast processing of positive boolean functions (PBF) performed by the FPP.

Another important feature of the FPPs is that they are able to perform binary morphological operations in parallel. That is, these operations are not performed by sequentially sliding a kernel over the binary image, but filtering over all neighbourhoods in the bit-plane at the same time. Therefore, traditional stack filters with a line structuring element can be extended into a morphological approach.

The paper is organized as follows: In section 2 the relationship between the threshold and bitwise decomposition is established. Section 3 shows a simple serial implementation to compute stack filters based on the relationship established in section 2. In section 4, the serial approach is extended to a bit-plane architecture suited for the FPP. Section 5 evaluates the hardware complexity of the algorithm. Finally, the conclusions of this work are stated in section 6.

2. FROM BITWISE TO THRESHOLD DECOMPOSITION

Let u_n be a discrete-time signal of index $n \in \mathbb{Z}$, with L levels, the binary threshold and bitwise decomposition are defined as in (1) and (2), respectively.

$$t_i(u_n) = \begin{cases} 1, u_n \geq i \\ 0, u_n < i \end{cases} \quad (1) \qquad b_k(u_n) = \left\lfloor \frac{u_n}{2^k} \right\rfloor \bmod 2 \quad (2)$$

where $i = 1, 2, ..., L$ and $k = 0, 1, ..., n_b - 1$ such that $n_b = \lceil log_2(L) \rceil$ represents the number of bits. The corresponding multilevel signals can be restored by using the stacking property (3), and the weighted stacking property (4).

$$u_n = \sum_{i=1}^{L} t_i(u_n) \quad (3) \qquad u_n = \sum_{k=0}^{n_b-1} b_k(u_n) 2^k \quad (4)$$

Since the bitwise decomposition is generated by a non-monotone increasing mapping, the threshold decomposition property does not hold. Therefore, non-linear filters cannot be applied over those binary signals straightforwardly. In order to overcome this problem, our algorithm regenerates threshold mappings from those obtained by the bitwise decomposition through simple boolean functions.

2.1. The Regenerative Boolean Function

According to [6], we can check whether u_n is equal ($h_l = 1$) or not ($h_l = 0$) to a level l by manipulating the bitwise decomposition of u_n as shown in (5).

$$h_l = \bigwedge_{k=0}^{n_b-1} \beta_k(l) \qquad \beta_k(l) = \begin{cases} b_k, & if\ w_k(l) = 1 \\ \overline{b_k}, & if\ w_k(l) = 0 \end{cases} \quad (5)$$

where $w_k(l) = \lfloor l/2^k \rfloor \bmod 2$. This test can be useful to regenerate a threshold decomposition mapping, since it implies to repeat

We gratefully acknowledge the valuable comments of Prof. Richard Kleihorst (Flemish Institute for Technological Research, Belgium).

iteratively this test for all levels in which any i-th threshold decomposition mapping is comprised. Consequently, the threshold decomposition can be expressed in terms of the bitwise decomposition of u_n, leading us to the *Regenerative Boolean Function* (RBF)

$$t_i(u_n) = \bigvee_{l=i}^{2^{n_b}-1} \bigwedge_{k=0}^{n_b-1} \beta_k(l) \tag{6}$$

It is worth to note that the RBF has the canonical form of a Sum of Products (SoP). Such a boolean expression can be minimized for every i-th mapping in order to reduce to a minimum the number of terms in the logical expression.

2.2. Minimization of the RBF

Boolean functions can be minimized by using a Karnaugh map (K-map). A K-map is a grid-like representation of a truth table, or equivalently, of a boolean expression. Boolean expression simplification with K-maps is basically performed by finding the largest *power-of-two* groups of *adjacent* cells containing the miniterms involved in the boolean expression. After grouping, simple boolean laws are applied to each group in order to eliminate redundancies and yield a minimal SoP expression.

In light of the adjacency and power-of-two grouping constraints imposed by the Karnaugh minimization, it seems logical to first of all analyse the K-maps resulting from multiple power-of-two threshold mappings.

2.2.1. *Minimization of the 2^{n_b-1}-th mapping*

The K-map of the 2^{n_b-1}-th mapping is presented in Fig. 1. Its boolean expression is $t_{2^{n_b-1}} = f(b_{n_{b-1}}b_{n_{b-2}}b_{n_{b-3}}...b_1b_0) = \sum m(2^{n_b-1}...2^{n_b}-1)$, where $\sum m(\cdot)$ represents the sum of miniterms involved in the expression. Since all miniterms are rectangular grouped in a power-of-two number, as outlined in red, the 2^{n_b-1}-mapping is minimized as shown in (7).

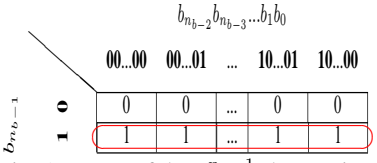

Fig. 1 K-map of the 2^{n_b-1}-th mapping.

$$t_{2^{n_b-1}} = b_{n_{b-1}} \tag{7}$$

2.2.2. *Minimization of the 2^{n_b-2}-th and $3\cdot 2^{n_b-2}$-th mapping*

The K-map of the 2^{n_b-2}-th and $3\cdot 2^{n_b-2}$-th mapping is presented in Fig. 2. Their corresponding boolean expressions are given by $t_{2^{n_b-2}} = \sum m(2^{n_b-2}...2^{n_b}-1)$ and $t_{3\cdot 2^{n_b-2}} = \sum m(3\cdot 2^{n_b-2}...2^{n_b}-1)$.

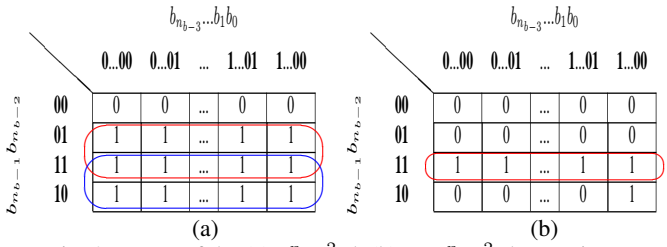

Fig. 2 K-map of the (a) 2^{n_b-2}-th (b) $3\cdot 2^{n_b-2}$-th mapping.

In Fig. 2(a), we can see that all miniterms can be grouped in two logical adjacencies, simplifying the corresponding RBF to (8). On the other hand, Fig. 2(b) leads to (9). Note that the $2\cdot 2^{n_b-2}$-th mapping corresponds to the 2^{n_b-1}-th one, which we already computed in section 2.2.1.

$$t_{2^{n_b-2}} = b_{n_{b-1}} \vee b_{n_{b-2}} \tag{8}$$

$$t_{3\cdot 2^{n_b-2}} = b_{n_{b-1}} \wedge b_{n_{b-2}} \tag{9}$$

2.2.3. *Mapping minimization of odd multiples of 2^{n_b-3}*

Similarly as in Sec. 2.2.2, we can find the minimal SoP expression of the $\lambda\cdot 2^{n_b-3}$-th mappings ($\lambda = 1,3,5,7$), by using their corresponding K-maps presented in Fig. 3.

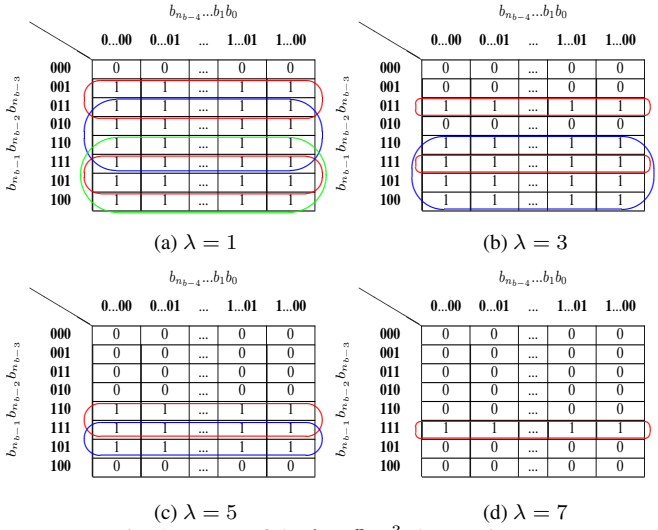

Fig. 3 K-map of the $\lambda\cdot 2^{n_b-3}$-th mappings

Fig. 3 shows the largest power-of-two logical adjacencies looped out for each K-map, which leads us to

$$t_{2^{n_b-3}} = b_{n_{b-1}} \vee (b_{n_{b-2}} \vee b_{n_{b-3}}) \tag{10}$$

$$t_{3\cdot 2^{n_b-3}} = b_{n_{b-1}} \vee (b_{n_{b-2}} \wedge b_{n_{b-3}}) \tag{11}$$

$$t_{5\cdot 2^{n_b-3}} = b_{n_{b-1}} \wedge (b_{n_{b-2}} \vee b_{n_{b-3}}) \tag{12}$$

$$t_{7\cdot 2^{n_b-3}} = b_{n_{b-1}} \wedge (b_{n_{b-2}} \wedge b_{n_{b-3}}) \tag{13}$$

In order to obtain the threshold mapping at any level expressed in bitwise mappings without appealing to its K-map, it would be necessary to know how many bits are required to regenerate the threshold mapping and the way to combine the $\vee$ and $\wedge$ operators. By carefully analyzing Figs. 1-3, we can realize that such information is actually coded in the binary representation of the i-th threshold level that we want to evaluate. For instance, in Fig. 2, we know that for any $u_n > 1$, the two most significant bits of the mappings $i = 2^{n_b-2}$ and $i = 3\cdot 2^{n_b-2}$ are $(01)_2$ and $(11)_2$, respectively. Therefore, in accordance with (8) and (9), we can establish that the two most significant bits of the decomposition of u_n are necessary to regenerate those threshold mappings. Furthermore, the corresponding boolean operator can be coded with the most significant bit, that is, $0 \rightarrow \vee$ and $1 \rightarrow \wedge$. Consequently, for the $i = \lambda\cdot 2^{n_b-2}$ ($\lambda = 1,3,5,7$) mappings depicted in Fig. 3, their *regeneration codes* correspond to $(001)_2, (011)_2, (101)_2, (111)_2$ telling us that three bits are necessary and the two most significant bits of the code provide the information about the way to combine the boolean operators. This can

be easily verified in (10)-(13); however, keep in mind that the order of combination of the boolean operators depends on the order of significance in the code.

Finally, we conclude that any i-th threshold mapping is expressed in the following generic and simplified form

$$t_i(u_n) = (b_{n_b-1} \Diamond_{d_{n_b-1}} (b_{n_b-2} \Diamond_{d_{n_b-2}} \cdots (b_{n_b-\ell+1} \Diamond_{d_{n_b-\ell+1}} b_{n_b-\ell}))) \quad (14)$$

where the binary representation of $(i)_{10} = (d_{n_b-1} d_{n_b-2} ... d_1 d_0)_2$ defines the *conditional boolean operator* presented below

$$\Diamond_x = \begin{cases} \vee, & if\ x = 0 \\ \wedge, & if\ x = 1 \end{cases} \quad (15)$$

On the other hand, ℓ represent the number of bits of the regeneration code, such that any threshold level i can be represented as $\lambda \cdot 2^{n_b-\ell}$. Where λ is an odd number that represents the regeneration code, while $2^{n_b-\ell}$ is an even number that depends on the number of bits in which λ is represented. For instance, assume that we want to get the threshold at $i = 168$; therefore, the regeneration code can be obtained as follows

$$(168)_{10} = (10101000)_2$$
$$\lambda \cdot 2^{n_b-\ell} = (21 \cdot 8)_{10} = (10101)_2 \cdot (1000)_2$$

Note that the regeneration code is formed by discarding the less significant bits set to zero of the binary representation of i. Therefore, ℓ can be defined as in (16), where $\#z$ represents the number of less significant bits set to zero.

$$\ell = n_b - \#z \quad (16)$$

3. BINARY SEARCH ALGORITHM BY THRESHOLD MAPPING REGENERATION

The Binary Search Algorithm (BSA) [7] is a stack filter implementation based on iteratively testing a threshold condition in the middle of the search space in which the T-th ranked element of the sequence is located. Thus, reducing the search space by half on each iteration. The threshold condition is generated by using the *binary-tree threshold*, which is defined as in (17).

$$L_1 = 2^{n_b-1} \quad for\ j = 1$$
$$L_j = \sum_{m=1}^{j-1} PBF(t_{L_m}(u_1), t_{L_m}(u_2), ..., t_{L_m}(u_n)) 2^{n_b-m} + 2^{n_b-j}, \quad for\ j \geq 2 \quad (17)$$

where PBF represents the positive boolean function of the non-linear filter. The binary search algorithm can be simplified by using the corresponding RBF to generate the binary-tree threshold (L_j) without performing its computation straightforwardly as in (17). In order to better show how the RBF can be used with the binary search algorithm, let us analyse the following example. Let $ROF_{(6)}\{6, 2, 11, 4, 8, 3, 14\}$ be the sequence from which we want to obtain the 6th ranked element. Table 1 better shows how the BSA works by using bitwise manipulations and the RBF.

The first step of the BSA (i.e. $j = 1$) consists of applying the binary-tree threshold in $L_1 = 8$ and the binary ROF_6 operator. Since the data sequence is bitwise decomposed with 4 bits, the MSB row exactly corresponds to the L_1 threshold. As a result, we rewrite directly this row into the 1st level of the binary-tree threshold column, while the 6-th ranked bit leads to 1. This output serves to compute the $L_2 = 12$ threshold when $j = 2$. This way, the binary-tree threshold is computed according to (14) and (15) leading us to

$$t_{L_2}(u_n) = b_3 \wedge b_2 \quad (18)$$

which implies that the binary-tree threshold when $j = 2$ is computed by applying the AND operator between the 1st and 2nd bit level. The result is shown in the binary-tree threshold column while the 6th ranked bit in the second level is 0. For the third level (i.e. $j = 3$) the binary-tree threshold is set to $L_3 = 10$ and it can be regenerated in terms of the bitwise decomposition by using the following RBF

$$t_{L_3}(u_n) = b_3 \wedge (b_2 \vee b_1) \quad (19)$$

In the binary-tree threshold column we can verify the result of this regeneration, and the 6th ranked bit at this level yields 1. Finally, for the fourth level, the threshold condition is $L_4 = 11$ and it is implemented with the following RBF

$$t_{L_4}(u_n) = b_3 \wedge (b_2 \vee (b_1 \wedge b_0) \quad (20)$$

The 6th ranked element is shown in the $ROF_{(6)}$ column, which is $u_3 = 11$.

		Data Sequence u_n							*Binary-tree threshold* $t_{L_j}(u_n)$							
j-level	b_k	6	2	11	4	8	3	14								$ROF_{(6)}$
1	b_3	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1
2	b_2	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0
3	b_1	1	1	1	0	0	1	1	0	0	1	0	0	0	1	1
4	b_0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	1

Table 1. Binary search algorithm.

It is worth to remark that by using RBFs to get the binary-tree thresholds rather than full comparators, the complexity of the BSA is reduced. Note that only basic logical manipulations were used, while the control flow of the algorithm is full binary encoded. In other words, the T-th ranked bit relates the boolean operator (i.e. $0 \rightarrow \vee$, $1 \rightarrow \wedge$) to be used to regenerate the thresholds, while the j-level of such bit relates the order of combination in which those operators should be applied. For instance, coming back to Table 1, we can see that the 6th ranked bit for $j = 1$ yielded 1, which implies to regenerate $t_{L_2}(u_n)$ by using $\wedge$ in (18). A similar process is observed for $t_{L_3}(u_n)$ since the two boolean operators used in (19) come from the regeneration code $(10)_2$, which is obtained from the ranked bits at $j = 1$ and $j = 2$. This sort of control flow codification of the BSA can be also verified for $t_{L_4}(u_n)$.

4. THE BIT-PLANE STACK FILTER ALGORITHM

The approach presented in the previous section can now be extended to a bit-plane architecture. Note that FPPs are able to perform PBFs and logical operations very efficiently at a bit-plane level. Therefore, we will apply the image bitwise decomposition to split up the multilevel image, and then use the BSA and threshold mapping regeneration to apply the respective non-linear filter.

Every bitwise decomposed bit-plane A_k is formed by $N = 2^{n_b-k-1}$ non-monotone mappings a_k^q as shown in (21), where k is the level of significance.

$$A_k = a_k^1 \vee a_k^2 \vee \ldots \vee a_k^N \qquad a_k^q = r_k^q \wedge \overline{s}_k^q \quad (21)$$

$r_k^q = t_{l_i}(I)$ represents the threshold mapping at level $l_i = (2q-1)\cdot 2^k$ of the multilevel image I. Similarly, $s_k^q = t_{l_f}(I)$ and $l_f = q \cdot 2^{k+1}$. Thus, r_k^q and s_k^q represent the initial and final threshold levels that compose each non-monotone mapping. The transformed bit-plane C_k is obtained after applying the non-linear filter ($\star$) to each threshold mapping with the flat structuring element se yielding

$$C_k = c_k^1 \vee c_k^2 \vee \ldots \vee c_k^N \qquad c_k^q = (r_k^q \star se) \wedge \overline{(s_k^q \star se)} \tag{22}$$

where r_k^q and s_k^q can be regenerated by using the RBF shown in (14).

Algorithm 1 is a top-down approach that processes each bit-plane from the most to the least significant level. In lines 1-3, the most significant bit-plane is processed directly since no further logic manipulation is required than applying the non-linear filter. On the other hand, in lines 4 and 5, the final threshold mappings to be used in the next level of significance depend on the initial thresholds computed in the actual level. Note that **0** represents a bit-plane of zeros.

From line 6 to 19 the rest of the levels of significance are processed, where from line 10 to 13, every c_k^q non-monotone mapping is computed. It is worth to remark that in lines 10 and 11 the threshold regeneration process takes place and its computation is full binary encoded. Finally, in lines 16 and 17, the final threshold mappings s_k^q are computed for the next level of significance.

Algorithm 1 Bit-plane stack filter algorithm

1: $r_{n_b-1}^1 \leftarrow A_{n_b-1}$
2: $f_{n_b-1}^1 \leftarrow r_{n_b-1}^1 \star se$
3: $C_{n_b-1} \leftarrow f_{n_b-1}^1$
4: $s_{n_b-2}^1 \leftarrow f_{n_b-1}^1$
5: $s_{n_b-2}^2 \leftarrow \mathbf{0}$
6: **for** $k = (n_b - 2)$ down to 0 **do**
7: $\quad C_k \leftarrow \mathbf{0}$
8: $\quad p = n_b - 2 - k$
9: $\quad$ **for** $q = 1$ to 2^{n_b-k-1} **do**
10: $\quad\quad (d_p d_{p-1} \ldots d_0)_2 \leftarrow (q-1)_{10}$
11: $\quad\quad r_k^q \leftarrow (A_{n_{b-1}} \Diamond_{d_p} (A_{n_{b-2}} \Diamond_{d_{p-1}} \ldots (A_{k+1} \Diamond_{d_0} A_k)))$
12: $\quad\quad f_k^q \leftarrow r_k^q \star se$
13: $\quad\quad c_k^q \leftarrow f_k^q \wedge \overline{s}_k^q$
14: $\quad\quad C_k \leftarrow C_k \vee c_k^q$
15: $\quad\quad m \leftarrow 2q - 1$
16: $\quad\quad s_{k-1}^m \leftarrow f_k^q$
17: $\quad\quad s_{k-1}^{m+1} \leftarrow s_k^q$
18: $\quad$ **end for**
19: **end for**

Some filters such as morphological operations and ROFs may have either extensive or anti-extensive properties. For instance, erosion shrinks bright regions while dilation expands them. These properties combined with the hierarchical processing of our algorithm may help to considerably reduce the processing time. In order to justify this statement, let's consider that ($\star$) is an anti-extensive filter in Algorithm 1. Therefore, we may conclude that

$$\textbf{if} \quad f_k^q = \mathbf{0} \qquad \textbf{then} \quad \forall f_{k-1}^{q'} : f_{k-1}^{q'} \subseteq f_k^q \Rightarrow f_{k-1}^{q'} = \mathbf{0} \tag{23}$$

This means that due to the anti-extensive properties of the non-linear filter, f_k^q in line 12 of Algorithm 1 may turn into a bit-plane of zeros. Consequently, all the initial thresholds filtered in the lower levels of significance and with greater threshold level will be also zero. In this way, we can avoid computing c_k^q mappings in the lower levels of significance, and thus reduce the processing time. Conversely, if ($\star$) is an extensive filter, f_k^q may turn into a bit-plane of ones. Therefore, all the final thresholds filtered in the lower levels of significance and with lower threshold level will be also one. Since the final thresholds are complemented, as shown in line 13, c_k^q turns to be zero and the processing time can be reduced. The dual expression of (23) is

$$\textbf{if} \quad f_k^q = \mathbf{1} \qquad \textbf{then} \quad \forall f_{k-1}^{q'} : f_k^q \subseteq f_{k-1}^{q'} \Rightarrow f_{k-1}^{q'} = \mathbf{1} \tag{24}$$

5. ALGORITHM COMPLEXITY

Several advantages can be remarked in terms of hardware complexity of the bit-plane stack filter algorithm. For instance, the multilevel image decomposition/recomposition only involves reading/writing on the bit-fields of each data word. Also, according to Algorithm 1, the most demanding operations are simple logical manipulations (i.e. $\vee$ and $\wedge$) and binary filters, both implemented at a bit-plane level. As mentioned previously, these operations are efficiently performed in FPPs. Therefore, implementing a stack filter of an image with an 8-bit depth and full dynamic range, lead us to perform 2046 logical operations and 255 binary rank filters at a bit-plane level.

6. CONCLUSIONS

In this work we presented a stack filter algorithm that fits to the architecture of a focal plane processor. Relying on the image bitwise decomposition to process the image in the binary domain, we were able to study in depth the relationship between bitwise and threshold decomposition. As a result, we found that the algorithm can be implemented by using simple logical operations and PBFs, while the control flow of the algorithm can be full binary encoded. Our approach outperforms the naive bit-plane implementation of the threshold decomposition since the thresholding and stacking steps are not computed by using comparators and full adders. Also, the processing time of our approach depends on the histogram distribution, which is a feature not found in other approaches.

7. REFERENCES

[1] P.D. Wendt, E.J. Coyle, and N.C. Gallagher, "Stack filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, pp. 898–911, August 1986.

[2] P. Soille, "On morphological operators based on rank filters," *Elsevier Journal of Pattern Recogn.*, vol. 35, pp. 527–535, 2002.

[3] D.Z. Gevorkian, K.O. Egiazarian, J.T. Astola, and O. Vainio, "Parallel algorithms and VLSI architectures for stack filtering using fibonacci p-codes," *IEEE Transactions on Signal Processing*, vol. 43, pp. 286–295, January 1995.

[4] C. Chakrabarti and L.E. Lucke, "VLSI architectures for weighted order statistic (WOS) filters," *Elsevier Signal Processing*, vol. 80, pp. 1419–1433, August 2000.

[5] M.J. Avedillo, J.M. Quintana, A. Hamid, and A. Jiménez-Calderón, "A practical parallel architecture for stacks filters," *Springer Journal of VLSI Signal Processing*, vol. 38, pp. 91–100, September 2004.

[6] A. Frías-Velázquez and J.R. Morros, "Histogram computation based on image bitwise decomposition," in *Proc. of IEEE Int. Conf. on Image Processing (ICIP'09)*, 2009, pp. 3259–3272.

[7] K. Chen, "Bit-serial realization of a class of nonlinear filters based on positive boolean functions," *IEEE Trans. Circuits and Systems*, vol. 36, pp. 785–794, June 1989.