VERY EFFICIENT IMPLEMENTATION OF MAX/MIN FILTERS

Dinu Coltuc and Philippe Bolon

LAMII-ESIA, University of Savoie,
41, Av. de la Plaine, P.O. BOX 806, 74016 Annecy, FRANCE
e-mail: (coltuc, bolon) @ esia.univ-savoie.fr

ABSTRACT

Implementation of fast and low-cost running max/min with only 3 comparator circuits is addressed. An algorithm of less than 3 comparisons per sample complexity is proposed and its implementation is discussed. The algorithm uses a data-block processing scheme where each window is split in two sub-windows. The maximum (minimum, respectively) in each sub-window is computed, step by step, by one comparison between an input sample and the maximum of an adjacent sub-window. Furthermore, final results are obtained by one more comparison each one.

1. INTRODUCTION

The interest for fast running max/min algorithms is due to their use in mathematical morphology (dilation and erosion of graylevel images are local max and min, respectively) as well as in certain low-level image and signal processing tasks. The fastest algorithms for max/min computation reported in the literature entails, regardless the size n of the filter window, the extremely low complexity of less than 3 comparisons per sample [1, 2]. They perform the computation on blocks of data and, for each block, only 2 output results are fully computed (step by step) from the input operands, while the other output results are immediately obtained by a single comparison between already available intermediate results. There is a difference in performance between these algorithms due to the selection of the data block size which yields a slightly lower complexity of $3 - 6/(n+1)$ [2] compared with $3 - 4/n$ in [1]. We mention that both algorithms are independent of data distribution and yield regular and periodic flowgraphs for any size of the filter window. Flowgraphs regularity and periodicity are of great interest when hardware implementations are in view. Besides, the computational complexity of such algorithms clearly suggests the possibility to derive, for any window size, fast architectures with only 3 comparison circuits.

Analyzing both algorithms, we have noticed that the computation flow is not perfectly regular. The processing for the final results is position dependent being computed either from already available partial results or, step by step, from the input operands. In order to eliminate this drawback, we propose a slightly modified algorithm. While still less than 3 comparisons per sample, i.e., $3-2/(n-1)$, the mathematical complexity of the new scheme is slightly lower than those of the above mentioned algorithms. On the other hand, the derived flowgraph yields a more efficient implementation. The paper presents the modified algorithm and discusses its implementation.

2. FAST ALGORITHM

Let $\{x_i, i = 1, 2, \ldots, \}$ be the input sequence. We shall refer only to the maximum computation; the minimum computation problem is equivalent. The output of the 1D running max filter within a window of size n is the sequence $\{y_i, i = 1, 2, \ldots\}$, where each y_i is the maximum of n consecutive samples:

$$y_i = \max(x_i, x_{i+1}, \ldots, x_{i+n-1}) \tag{1}$$

The basic idea of the algorithm is to obtain sequences of partial results, namely maximum values on sub-windows, such that they can be further used for the very fast computation of $n-1$ output results. Before giving the general algorithm, we briefly describe the computation scheme for the particular case of a window $n = 5$.

The output sequence, when $n = 5$, is:

$$\begin{aligned} y_1 &= \max(x_1, x_2, x_3, x_4, x_5) \\ y_2 &= \max(x_2, x_3, x_4, x_5, x_6) \\ y_3 &= \max(x_3, x_4, x_5, x_6, x_7) \\ y_4 &= \max(x_4, x_5, x_6, x_7, x_8) \\ y_5 &= \max(x_5, x_6, x_7, x_8, x_9) \\ y_6 &= \max(x_6, x_7, x_8, x_9, x_{10}) \\ &\vdots \end{aligned} \tag{2}$$

The straight computation of the maximum value of 5 samples needs 4 comparisons [3]. Since, for consecutive output values, 4 operands out of 5 are common, the idea of computation flow optimization in order to take advantage of common operands naturally appears. Thus, it is expected that the complexity of the running maximum is considerably lower than that of a single maximum computation.

Let be the first 4 output results computed as follows:

$$\begin{aligned} y_1 &= \max(\max(x_1, x_2, x_3, x_4), x_5) \\ y_2 &= \max(\max(x_2, x_3, x_4), \max(x_5, x_6)) \\ y_3 &= \max(\max(x_3, x_4), \max(x_5, x_6, x_7)) \\ y_4 &= \max(x_4, \max(x_5, x_6, x_7, x_8)) \end{aligned} \tag{3}$$

The next step is to observe that the computation of the inner operands can be further detailed to take advantage of common partial results. Thus, the first operand which appears in the computation of y_1 can be evaluated as:

$$\max(x_1, x_2, x_3, x_4) = \max(x_1, \max(x_2, x_3, x_4)) \tag{4}$$

by means of a single comparison between an input sample (x_1) and an operand which appears in the computation of y_2. Furthermore:

$$\max(x_2, x_3, x_4) = \max(x_2, \max(x_3, x_4)) \tag{5}$$

which uses $\max(x_3, x_4)$, an operand which appears in the computation of y_3 and whose computation needs only one comparison.

Similarly, the second operands in (3) can be evaluated step by step, with only 3 comparisons, such that, each step, an input sample is compared against a partial result and the maximum is propagated. Thus, the partial results are: $\max(x_5, x_6)$, $\max(\max(x_5, x_6), x_7)$, $\max(\max(x_5, x_6, x_7), x_8)$.

Remark: There is a difference between the computation of the inner operands in (3); the inner left operands are computed in the reverse order of the index of the input samples, namely: x_4, x_3, x_2, x_1, while the inner right ones in the normal order: x_5, x_6, x_7, x_8.

After the first 4 output results are obtained, the next group of 4 are computed and, so on. The computational complexity of the proposed scheme follows by counting the number of comparisons for the computation of 4 output results, namely 4 comparisons to obtain the final results and two times 3 comparisons to get the partial results. This yields a computational complexity of 2.5 comparisons per sample.

2.1. The general case

The algorithm considers the computation of groups of $n-1$ results and starts the processing with $2n-2$ input samples. The data block is split in two equal groups of consecutive samples, namely $\{x_1, x_2, \ldots, x_{n-1}\}$ and $\{x_n, x_{n+1}, \ldots, x_{2n-2}\}$.

For the group of partial results computed in descending order, one starts from tail to head with x_{n-1}, the last sample in the group. Each step i, $i = 1, \ldots, n-1$, a comparison is done between an input sample and the maximum found in the previous step. We denote each result, i.e., the maximum of a consecutive group of samples, by $d_{k,j}$, where the subscripts k and j are the first and the last index in the group. Since for the first group of $n-1$ samples $j = n-1$, one has:

$$d_{k,n-1} = \max(x_k, x_{k+1}, x_{k+2}, \ldots, x_{n-1}) \tag{6}$$

The first subscript k is related to the step i of the computation (in each group) by $k = n - i$. Thus, the first result of the sequence is:

$$d_{n-1,n-1} = x_{n-1} \tag{7}$$

For $i = 2, \ldots, n-1$, one has:

$$d_{n-i,n-1} = \max(x_i, d_{n-i-1,n-1}) \tag{8}$$

The complete sequence of sub-windows is processed one by one doing a single comparison per result.

For the second group (ascending order), the computation is performed from head to tail starting with x_n, the first sample in the group. Let the partial results be $a_{n,q}$. The sequence of partial results follows by taking:

$$a_{n,n+i-1} = \max(a_{n,n+i-2}, x_{n+i-1}) \tag{9}$$

where, for $i = 1$, the first partial result is:

$$a_{n,n} = x_n \tag{10}$$

As above, each result in the sequence is:

$$a_{n,n+i-1} = \max(x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+i-1}) \tag{11}$$

where $1 \leq i \leq 2n-2$. Obviously, each step demands only one comparison.

Next, the output sequence immediately appears as:

$$y_i = \max(d_{i,n-1}, a_{n,n+i-1}) \tag{12}$$

Otherwise stated, the final results are obtained with only one comparison between partial results.

The computation continues periodically for groups of $n-1$ results by finding, as above, $d_{i,2n-1}$, $a_{2n,q}$ and so on.

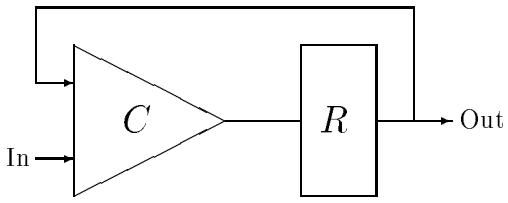

Figure 1: Partial results computation; block diagram.

The computational complexity of the modified algorithm follows by counting the number of comparisons/sample needed for each group of $n-1$ output values. Thus, $n-2$ comparisons are needed to compute $d_{i,j}$ terms, other $n-2$ comparisons for $d_{p,q}$ terms and, finally, $n-1$ comparisons for the output results. Therefore, we have:

$$C(n) = \frac{3n-5}{n-1} = 3 - \frac{2}{n-1} \qquad (13)$$

As it can be seen, in terms of computational complexity, the algorithms reported in [1, 2] slightly outperforms the modified algorithm. The difference is due to the size of data-block considered in the computation.

3. IMPLEMENTATION

The processing is naturally split in 3 parts: 1) computation of partial results (ascending order); 2) computation of partial results (descending order); 3) computation of final results.

3.1. Computation of partial results

The computation of each sequence of partial results can be implemented by a simple closed loops scheme [4, 5]. The input operands are delivered, each clock cycle, one by one, on the data input In (Fig. 1). The C block delivers at the output the maximum of the input operands. The R block is simply a feed-back register. The C block can be implemented by a multiplexer whose output is selected by a comparator.

Let us consider the computation of the first sequence of $d_{i,j}$ results. We suppose that the feed-back register is set to "0" prior to the first clock. At the rising edge of the first clock cycle, x_{n-1} is at input; at the output of the comparator, one has the result of the comparison between x_{n-1} and "0", i.e., x_{n-1}, result which is stored, on the falling edge of the clock tick, in register R. At the rising edge of the second clock tick, x_{n-2} is input. Now, the output of the comparator has the result of the comparison between

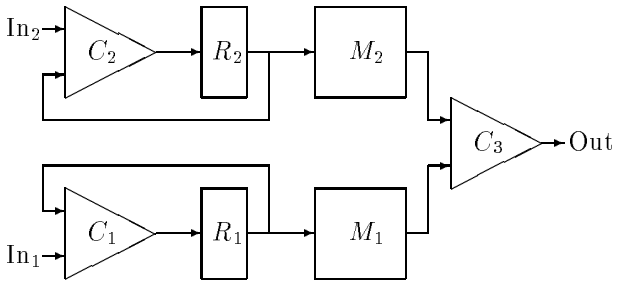

Figure 2: Running max/min architecture.

x_{n-2} and the content of the feed-back register (x_{n-1}), namely $\max(x_{n-1}, x_{n-2})$, result which over-writes the feed-back register. Thus, after $n-1$ clock ticks, the feed-back register contains $\max(x_1, x_2, \ldots, x_{n-1})$.

To obtain the partial results for the next group of samples, the procedure goes on identically, taking care to set to "0" the feed-back register before the first comparison.

The scheme computes the partial results in ascending or descending order, depending on the order of the sequence of input operands.

3.2. Computation of final results

Each final result requires only one comparison between two already computed partial results. Thus, the final results are obtained by a C block feed-on with the partial results given in the appropriate order.

The two sequences of partial results (ascending and descending order) can be computed in parallel. Therefore, the algorithm can be implemented by a two-stage scheme: one stage for the computation of partial results and the other for the computation of final results. A block diagram of the proposed architecture is shown in Fig. 2. Thus if the input operands are provided in the appropriate order, C_1 delivers each clock cycle a $d_{i,j}$ result, while C_2 an $a_{p,q}$ one. Furthermore, the partial results are stored in the memory buffers M_1 and M_2. The size of the memory buffers is of $n-1$ samples each one. Feedback registers should be cleared each $n-1$ clock cycles. The output result is computed by C_3 which receives the appropriate inputs from M_1 and M_2. Data dependencies and the flow of operands for a window of size 5 is presented in Table 1.

The algorithm and, consequently, its implementation refers to 1D running max/min. The straight extension to the 2D case does not hold since, for consecutive window positions, there are more than one different samples which appear. For instance, in the case of

Table 1: Snapshot of data flow and dependence ($n = 5$).

Clock	In_1	In_2	R_1	R_2	M_1	M_2	Output
1	x_4	x_5	0	0	$d_{4,4}$	$a_{5,5}$	
2	x_3	x_6			$d_{3,4}$	$a_{5,6}$	
3	x_2	x_7			$d_{2,4}$	$a_{5,7}$	
4	x_1	x_8			$d_{1,4}$	$a_{5,8}$	
5	x_8	x_9	0	0	$d_{8,8}$	$a_{9,9}$	
6	x_7	x_{10}			$d_{7,8} \rightarrow d_{1,4}$	$a_{9,10} \rightarrow a_{5,5}$	$y_1 = \max(d_{1,4}, a_{5,5})$
7	x_6	x_{11}			$d_{6,8} \rightarrow d_{2,4}$	$a_{9,11} \rightarrow a_{5,6}$	$y_2 = \max(d_{2,4}, a_{5,6})$
8	x_5	x_{12}			$d_{5,8} \rightarrow d_{3,4}$	$a_{9,12} \rightarrow a_{5,7}$	$y_3 = \max(d_{3,4}, a_{5,7})$
9	x_{12}	x_{13}	0	0	$d_{12,12} \rightarrow d_{4,4}$	$a_{13,13} \rightarrow a_{5,8}$	$y_4 = \max(d_{4,4}, a_{5,8})$
10	x_{11}	x_{14}			$d_{11,12} \rightarrow d_{5,8}$	$a_{13,14} \rightarrow a_{9,9}$	$y_5 = \max(d_{5,8}, a_{9,9})$
11	x_{10}	x_{15}			$d_{10,12} \rightarrow d_{6,8}$	$a_{13,15} \rightarrow a_{9,10}$	$y_6 = \max(d_{6,8}, a_{9,10})$
12	x_9	x_{16}			$d_{9,12} \rightarrow d_{7,8}$	$a_{13,16} \rightarrow a_{9,11}$	$y_7 = \max(d_{7,8}, a_{9,11})$
13	x_{16}	x_{17}	0	0	$d_{16,16} \rightarrow d_{8,8}$	$a_{17,17} \rightarrow a_{9,12}$	$y_8 = \max(d_{8,8}, a_{9,12})$

2D running max/min in a $n \times n$ rectangular window, n samples (one column) change for adjacent windows (on row direction). A 2D approach by passing in n steps from one window to another has been discussed in [5]; its major drawback is the severe loss in performance since only one result out of n computed ones is valid. The solution, in most 2D cases, is to take advantage of separable windows (and the rectangular ones are) by filtering in two passes, an 1D filtering on rows followed by an 1D filtering on columns.

3.3. Variable window size

A hardware processor for running max/min for a fixed window size is obviously of very limited interest since the window size is a flexible parameter in filtering as well as in mathematical morphology.

The extension of the implementation for a range of window sizes is straightforward. The architecture does not change, i.e., regardless the window size, one needs exactly 3 comparison processing units. Changes will be in the capacity of buffer memories (M_1 and M_2) including addressing logic and timing. Thus, the buffers should be designed for the maximum window size ($N - 1$ memory cells for a window size N). For any window $n \leq N$, the processor works as described above by changing only the data addressing and the command timing. While the output rate of the processor is constant, obviously, the latency depends on n (the first result is obtained after $n + 1$ clock cycles).

4. CONCLUSIONS

A fast low-cost implementation for running max/min filters using only 3 comparators regardless the window size has been presented. The implementation is derived from a very fast algorithm of less than 3 comparisons per sample complexity. The fast algorithm is optimized to yield a balanced flow of operations. Implementation for programmable window sizes follows immediately by changing only the timing diagram and the addressing logic.

5. REFERENCES

[1] J. Gil, M. Werman, "Computing 2-D Min, Median and Max Filters"", *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 15, no. 5, pp. 504-507, 1993.

[2] D. Coltuc, I. Brulea, V. Buzuloiu, "A Very Fast Algorithm for Max/Min Filtering"", *IEEE Intl. Conference on Electronics, Circuits and Systems, ICECS'96*, Rhodos, Greece, vol. I, pp. 464-466, 1996.

[3] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

[4] I. Pitas, "Fast Algorithms for Running Ordering and Max/Min Calculation", *IEEE Trans. on Circuits and Systems*, vol. 36, no. 6, pp. 795-804, 1989.

[5] D. Coltuc, I. Pitas, "Fast Computation of a Class of Running Filters", *IEEE Trans. on Signal Processing*, vol. 46, no. 3, 549-553, 1998.