Median architecture by accumulative parallel counters

Median Filter Architecture by Accumulative Parallel Counters

J. O. Cadenas, G. M. Megson, and R. S. Sherratt, *Fellow, IEEE*

***Abstract*—The time to process each of *W*/*B* processing blocks of a median calculation method on a set of *N* *W*-bit integers is improved here by a factor of three compared to the literature. Parallelism uncovered in blocks containing *B*-bit slices are exploited by independent accumulative parallel counters so that the median is calculated faster than any known previous method for any *N*, *W* values. The improvements to the method are discussed in the context of calculating the median for a moving set of *N* integers for which a pipelined architecture is developed. An extra benefit of smaller area for the architecture is also reported.**

***Index Terms*—Median Filter, Pipelined Processing, Image Processing.**

I. Introduction

THE selection, and in particular the calculation of the median arises in many applications in computer science and statistics that extends to a variety of fields [1, 2]. The median, *M*, of a set of integers is such that half the integers in the set are less or equal to *M*, and half are greater or equal to *M*. For *N* sorted integers, the median is the integer at position $P = \lceil N/2 \rceil$ or the middle position. The median filter is essentially the computation of the median for a moving set of *N* values within a window.

The median can be calculated by sequential or parallel methods, without performing a full sort (that would take $O(n\log n)$ time), with a complexity of $O(n)$ [1], although these methods are not hardware amenable. Non-sorting based methods, especially those designed for hardware architectures achieve the calculation in a number of steps related to the bit length of unsigned integers, *W*, rather than *N*. Of these, probably the best method compared to previous sorting and non-sorting methods takes *W* processing steps to find the median [3]. Lately, two architectures have been proposed using the sorting method [4, 5]; in these the number of processing blocks are related to *N* which intuitively seem area-costly for the common case of $N > W$. This paper improves our previous method to calculate the median on a set of *N* *W*-bit integers in *W*/*B* processing blocks, where *B* is a parameter of how many bits are blocked for processing [6]. Each block contributes *B*-bit towards finding the median. A designer has to analyze the tradeoffs of parameters *N*, B and *W* in order to produce a winning architecture. For instance, our previous architecture [6] is made faster than the work in [3] only for $N > 7$ when working on slices of $B = 2$, 3 or 4 bits. The improvement here makes the method faster than previous work [6] for any *N* while maintaining blocks of 2-bit or 3-bit for practical hardware implementations. In fact, an analysis indicates the architecture presented here is faster than previously found even in the case of 1-bit slices ($B = 1$). As each block contributes *B*-bit to the median, the key idea in this paper is to maintain a parallel accumulation to select these *B* bits within each block, whereas previously, this accumulation was computed serially within a block. The novel approach that led to the improvement in this work relies on the concept of Accumulative Parallel Counters (APC) [7]. This paper starts by applying the APC concept to a set of *N* non-negative integers (or single window) using a small value of *N* as an example. APC is then applied to the case of maintaining the accumulation on a sliding window of size *N*, from where an architecture for calculating the median follows.

II. Accumulative Parallel Counters

APC is defined as an *l*-bit register that is updated by the sum of the previous contents and its *r* 1-bit inputs [7]. For instance, for a 3-bit register with a current value of 3 and four 1-bit input vector values of [0,1,1,0], the register value is incremented by 2 and thus updated to 5. This can be considered as if the number of ones in the 1-bit input were accumulated. An APC circuit with *r* 1-bit inputs is arranged in such a way that the delay to perform its operation in terms of full/half adders using a *l*-bit ripple carry adder is given by $\lfloor \log_2 r \rfloor + l$; details in [7]. The impact that this result has for the median architecture in this paper will be discussed later in the timing analysis of Section V.

Our median calculation method slices each *W*-bit data item by *B*-bit to arrange for *W*/*B* processing blocks. Within each block, accumulation of slices of bits are kept using an array of APC registers. Consequently, within a block, a number of 2^B APC registers are maintained; the first one being of $r = 1$ 1-bit input, the second of $r = 2$ 1-bit inputs, and the last one being of $r = 2^B$

J. O. Cadenas and R. S. Sherratt are with the School of Systems Engineering, The University of Reading, Reading RG6 6AX, UK (e-mail: o.cadenas@reading.ac.uk, sherratt@ieee.org).

G. M. Megson is with University of Westminster, London W1T 3UW, UK (e-mail: g.megson@westminster.ac.uk).

1-bit inputs. In general, given q_i of $r = 2^B$ 1-bit input then an array of 2^B APCs, $A_i = \sum_{i=0}^{r-1} q_i$ is arranged for processing per block. For N data items within a window, each APC register A_i is of $l = \log_2 N$ bits. Before an example is presented, it is worth recalling how to generate a 1-bit input vector of length $r = 2^B$ taken B-bit slices from data items.

A. *Generation of bit vectors*

An item data bit is interpreted as having disjoint amplitudes a_0 and a_1 for bit values 0 and 1 respectively. The item data bit is manipulated to be expressed in the form $d = a_0 Q_{[0]} + a_1 Q_{[1]}$ defining $Q_{[0]} = [0\ 1]$ and $Q_{[1]} = [1\ 0]$ so that when a data bit value is 0 it is represented as $Q_{[0]}$ or as $Q_{[1]}$ otherwise. This expression is based on quantum representations of bits [8], so let's call $Q_{[d]}$ a qubit. Operations on qubits such as a tensor between two or multiple qubits can now be defined. For instance, the tensor between two qubits is defined as:

$$Q_{[g]} \times Q_{[h]} = [g_1 \quad g_0] \times [h_1 \quad h_0] = [g_1 h_1 \quad g_1 h_0 \quad g_0 h_1 \quad g_0 h_0]$$

Thus, for two bits $x_0 x_1 = 10_2$, the qubit tensor is $Q_{[x_1 x_0]} = [1 \quad 0] \times [0 \quad 1] = [0 \quad 1 \quad 0 \quad 0]$. The tensor between two qubits is already familiar to us; it is equivalent to a binary decoding on two bits; a 2-to-4 binary decoder. The method here manipulates bit slices of data input as qubits and build its tensor; on a B-bit slice this is equivalent to performing a binary decoding operation of B bits to generate a 2^B bit vector; a B-to-2^B binary decoder [9]. This bit vector is the one previously referred to as vector q with size $r = 2^B$. For the specific case of $B = 2$, $q = [q_3\ q_2\ q_1\ q_0]$ and $r = 4$. APC register A_0 takes as input q_0, A_1 takes as input $q_1 q_0$, A_2 takes as input $q_2 q_1 q_0$ and A_3 takes as input $q_3 q_2 q_1 q_0$. A median calculation procedure using APC registers proceeds as in the following example.

B. *Small Example*

Consider a data set of $N = 9$ integers, $x_j = \{3, 1, 29, 21, 16, 9, 11, 19, 17\}$, each of $W = 5$ bits (labelled as [4:0]). Note $P = \lceil N/2 \rceil = 5$. Using all the five bits representation of each integer (and applying a qubit tensor) requires a full 5-to-32 binary decoder generating a bit vector of length $r = 2^5 = 32$ bits. Performing a 5-to-32 binary decoding for each integer in the set (and OR-ing into a bit vector of size 32, with all 32-bit positions initially in zero), generate the bit mapping presented in Table I. The binary decoding produces an indirect ordering of the integers in the set and then the median can be taken directly as 16_{10} since it is the middle position of the nine ones in the 32-bit vector q (or the $P = 5$ position of the nine ones in the vector). However, as the input size W grows in bits and other nuisances (such as repeated integers in the set) make this full binary decoding approach impractical to be used as a direct method for computing the median at least for sizes $W > 8$ [10]. Nevertheless the approach can be used as the principle of operation when processing slices of bits taken from the input integers instead of taking all bits at once.

For illustration, let us partition each $x_j[4:0]$ into two blocks of bits as $\{x_j[4:2], x_j[1:0]\}$; that is a first block with $B = 3$ bits (Block 1) and a second block with $B = 2$ bits (Block 2). For Block 1, eight ($2^B = 2^3 = 8$) APC registers A_i are maintained, $i = 0, \ldots, 7$ (and each wide enough to accumulate a count of up to $N = 9$). The three-bit slices on all $x_j[4:2]$ are processed first by Block 1 and then $x_j[1:0]$ by Block 2 as shown in Table II (note the dot in $(x_j)_2$). Integers are processed one at a time, and a binary decoding of the integer slice is performed on the fly into a 2^B bit q vector (for example slice "000" of integer 3 in Block 1 is decoded as vector $q = [0,0,0,0,0,0,0,1]$). From this decoded bit vector the input to a given APC is selected as previously stated. For instance, register A_2 has as input three 1-bit taken from the decoding vector the bits $q_2 q_1 q_0$ = "001", thus A_2 register will update to a count of 1. Therefore, in general, APC register A_i operates on i+1 1-bit inputs taken from the decoding vector q all bits with indices $0, \ldots, i$. All APC registers are updated in parallel for each integer slice as shown in Table II. The running count is shown on each APC after the slice for each input integer is processed. After all nine integers are processed by Block 1 A_7, A_6, … A_1, A_0 have counts 9, 8, 8, 7, 4, 4, 2, 2 respectively. This is the count after the slice for integer 17_{10} (last in input window) is processed.

TABLE II
MEDIAN CALCULATION PROCEDURE FOR NINE INTEGERS OF 5 BITS EACH. BLOCK 1 PROCESSES THREE BITS AND BLOCK 2 TWO BITS

Window Set		*Block 1 (3 bits)*								*Block 2*			
$(x_j)_{10}$	$(x_j)_2$	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	A_3	A_2	A_1	A_0
3	000.11	1	1	1	1	1	1	1	1	0	0	0	0
1	000.01	2	2	2	2	2	2	2	2	0	0	0	0
29	111.01	3	2	2	2	2	2	2	2	0	0	0	0
21	101.01	4	3	3	2	2	2	2	2	0	0	0	0
16	100.00	5	4	4	3	2	2	2	2	1	1	1	1
9	010.01	6	5	5	4	3	3	2	2	1	1	1	1
11	010.11	7	6	6	5	4	4	2	2	1	1	1	1
19	100.11	8	7	7	6	4	4	2	2	2	1	1	1
17	100.01	9	8	8	7	4	4	2	2	3	2	2	1
		1	1	1	1	0	0	0	0	1	1	1	1
		P = 5; $A_i \geq P$								*P = 1; $A_i \geq P$*			

C. Finishing the example: calculating the median

Calculating the median proceeds in a similar way to the procedure shown in our previous work [6]. A given block finds *B*-bit of the median as the first occurrence of the index *i* (right to left in Table II) for when $A_i \geq P$; this comparison is parallel. For Block 1 this comparison resolves to the bit vector "11110000" using $P = 5$ (shown at the bottom of Table II). Applying a priority encoding [9] to this vector (with priority right to left) gives the index $i = 4$ which corresponds to the column under APC A_4. This index corresponds to slices of three bits with values "100". Thus, the three MSBs of the median, are found as $M[4:2]$ = "100". From the nine input integers in the window, only integers 16, 19 and 17 had their processed bit slices with values "100" indicating that only the integers 16, 19 and 17 are still median candidates (highlighted in gray in Table II). Integers 3, 1, 29, 21, 9 and 11 need to get nullified so they cannot update any A_i for Block 2.

Next, Block 2 is processed. First, position for median P is recalculated as $P = 5 - 4 = 1$ (4 being the A value to the right under A_4 column, underlined in Table II for Block 1). Computing A_i proceeds as before, on the remaining 2-bit slice for all x_j. The condition $A \geq P$ is first satisfied for A under $i = 0$. The remaining two bits for the median are thus $M[1:0]$ = "00". Concatenating the results from blocks 1 and 2 give the median as $M = 10000_2 = 16_{10}$.

D. Key Observations for Improvements

From the above example, the following key observations are made as regard to the improvements to the method presented here. Firstly, reformulating the accumulation of bits in terms of APCs makes A_i to be computed in parallel and as decisions for finding the median are made on parallel logic decisions on accumulations, A_i, the method should be faster than the method as it stands [6]. The previous method was equivalent to calculating the histogram on slice values (in parallel) and then accumulating the histogram right to left (a serial process); this is one key difference. Secondly, the nullification of integers, that are not candidates for the median, is easier to handle when postponed until the next block. This leads to the third observation; further optimizations can be made to each APC arrangement for the case of a sliding window of *N* integers accepting a single integer. This observation is valid for the front-end processing block (the one that processes slices of the MSB bits). In this case a single integer leaves the window while a new integer arrives into the window. The front-end block sees and discards at most one integer within a window which can be conveniently exploited into improvements as presented next.

TABLE I
BIT MAPPING FOR THE PRESENTED EXAMPLE

q_{31}	q_{30}	q_{29}	q_{28}	q_{27}	q_{26}	q_{25}	q_{24}	q_{23}	q_{22}	q_{21}	q_{20}	q_{19}	q_{18}	q_{17}	q_{16}	q_{15}	q_{14}	q_{13}	q_{12}	q_{11}	q_{10}	q_9	q_8	q_7	q_6	q_5	q_4	q_3	q_2	q_1	q_0
0	0	1	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0

III. APCs ON A SLIDING WINDOW

Consider a continuous streaming of input integers arriving one at a time for processing; a median filter is interested in finding the median on the most recent *N* integers, and so we have a running window of size *N*. Once a pipeline with *N* integers gets full, a single old integer leaves the window while a single new integer arrives into the window. For the method here, a processing mechanism requires a coherent update on accumulations A_i for a correct fully streaming pipelined operation. Such an update can be thought of as a parallel subtraction of the contribution of the oldest integer slice and likewise an addition of the newest integer slice contribution. Consider a stream of integers as x_j = {3, 1, 29, 21, 16, 9, 11, 19, 17, 14, …,}. The first window of nine integers is the one presented in Table II. The second window is now composed of integers {1, 29, 21, 16, 9, 11, 19, 17, 14}; the oldest integer in the window was of value 3 and a new integer of value 14 enters the window. With the new window, repeating the whole computation of A_i for Block 1 in Table II gives counts of [9, 8, 8, 7, 4, 3, 1, 1]. The question is, given that the previous window had a count of A^{t-1} and knowing the new value x^t_j (value 14) and the old value x^{t-N}_j (value 3), can count A^t be computed? The slice for old integer value 3 ("00011", $i_{old} = 0$) is first decoded as q = [0,0,0,0,0,0,0,1] and to keep A_i coherent requires subtracting [1,1,1,1,1,1,1,1] to the running accumulation of [9,8,8,7,4,4,2,2] of the previous window (bottom of Table II, Block 1). The slice for integer value 14 ("01110", $i_{new} = 3$) is decoded as q = [0,0,0,0,1,0,0,0] and to keep A_i coherent requires adding [1,1,1,1,1,0,0,0] to the running accumulation. The net effect is subtracting (in parallel) the vector value [0,0,0,0,0,1,1,1] ([1,1,1,1,1,1,1,1] XOR [1,1,1,1,1,0,0,0] = [0,0,0,0,0,1,1,1]) from the running accumulation; that is [9,8,8,7,4,4,2,2]]-[0,0,0,0,0,1,1,1] = [9, 8, 8, 7, 4, 3, 1, 1]. Thus, A_i is updated from [9,8,8,7,4,4,2,2] (for window {3, 1, 29, 21, 16, 9, 11, 19, 17}) to [9, 8, 8, 7, 4, 3, 1, 1] (for window {1, 29, 21, 16, 9, 11, 19, 17, 14}).

A. Update Logic on APCs

Note the following from the discussion above: Firstly, the slice decoding process sets a bit *i* in the decoded vector *q* and then all the bits *i*+1, $i = 0, \ldots, 2^B-1$, are also set before being added or subtracted. This is in effect a sign-bit extension for a vector of length 2^B. Let's denote the sign extension on the decoded vector by $sign(q_i)$; with bit vectors size of 2^B bits. Secondly, the sign-extended decoded values of old and new slices are XOR-ed. A full analysis of what has to be performed to maintain a coherent accumulation A_i is given by:

$v = sign(q_{i\_old})$ XOR $sign(q_{i\_new})$

if (i_old < i_new) **then** $A_i = A_i - v$
else $A_i = A_i + v$

Fig. 1 shows the internal architecture of a front-end median processing block similar to Block 1 of Table II for $B = 2$. The block accepts integer inputs x_j and median position for this block P_{in}; it is most convenient to accept input M_{in} holding the median slice value found by a previous block. The median slice found by this block is generated at the bottom as M_{out} as well as the median position to be used by a next block P_{out}. Fig. 1 processes 2-bit slices of a window of $N = 5$ integers x_j of W-bit each and thus an array of four accumulative parallel counter registers of up to four 1-bit inputs with each register of size 3 bits is arranged. The sign extension performed on the decoder outputs can be maintained in parallel using gates with a fan-in of at most 2^B inputs. Alternatively, a binary-to- thermometer encoding can replace the decoding and sign-extension for a direct look-up table implementation [11]. Also, notice the decoders can be inhibited by a single enable bit to account for nullification of integers within a running window; they are fully enabled for the front-end block by simply making $M_{in}[B\text{-}1{:}0] = x_j[W\text{-}1{:}W\text{-}B]$. This is the reason to have *Min* as input, to contain nullification signals within a block rather than passing these from one block to the next (that would require N bits). A full circuit arrangement for the APC [7] is not necessary due to the fact that at most a single integer arrives or leaves the window; this is a key further optimization to a front-end block as of Fig. 1. After the comparison $A_i \geq P$ is performed (comparator block in Fig. 1) a priority encoder produces the median slice for this block. A simple arrangement of a (priority) multiplexer acting on the comparison output to select the value to be subtracted from input P_{in} to generate output P_{out} completes the operation of the block (Multiplexer/Adder block in Fig. 1).

Note the pipeline arrangement is clear from Fig. 1; a delay of N clock cycles is required to see all integers from a window (left to right registers in Fig. 1) plus the extra delay top to bottom in the architecture of Fig. 1; this delay is denoted by L. The gray boxes in Fig. 1 are to indicate places where extra registers might be necessary for faster pipelined operation. The overall latency for a front-end block is of $N+L$ clock cycles, after this latency a median slice is produced every clock cycle.

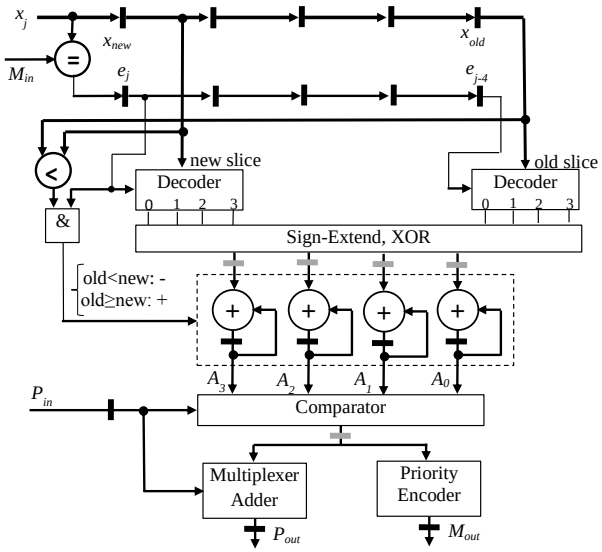

Fig. 1. A front-end block processing 2-bit slices on 5 integers. This block generates 2-bit of the median M_{out} and the P value (P_{out}) for the next block.

IV. Median Architecture for a Sliding Window

In general, for W-bit integers, physical blocks of B-bits each arrange for $\lceil W/B \rceil$ processing blocks; however it is possible to make each processing block to operate with its own B value (of bits) as shown from Table II. Fig. 2 shows the block diagram for computing the median for the example in Table II. Front-end block (Block 1) computes the 3-bit median, $M[4{:}2]$, and a second processing block (Block 2) computes the remaining 2-bit median $M[1{:}0]$ as detailed in Table II. $M[4{:}2]$ is available after $N+L$ clock cycles; thus the next processing block needs to get aligned in time by delaying input x_j by L clock cycles so the current input window is already loaded into the next processing block. Note for the next processing block, the assumption made earlier of having a single old integer leaving the window while a single new integer arrives into the window, is no longer valid. Consequently the simplifications seen in Fig. 1 cannot be used directly. Fortunately, the processing block previously presented in [6] can be used instead except with two key modifications. The first one is that integer nullification is replaced by exactly the scheme of Fig. 1 here. So, $M[4{:}2]$ computed by the front-end block gets compared to $x_j[4{:}2]$ seen by the processing block. The

second modification is that full APC circuit arrangements as detailed in [7] can be incorporated into the block for a parallel accumulation. This is so since more than one integer can enter or leave within a window for Block 2 as it is clearly shown in Table II. Each APC accumulator (there are 2^B) has an N-bit input vector with a register output of length $\log_2 N$. Results from a processing block are pipelined vertically where the calculation continues to the next block concatenating the generated median bit slices as shown in Fig. 2; the median emerges every clock cycle after an initial latency of $N+2L$ clock cycles in Fig. 2. This result is consistent with latest methods of $O(1)$ time for calculating running medians [12]. In a generic case of K processing blocks ($K = \lceil W/B \rceil$ if B-bit are processed by each block), the median M is found every clock cycle with a latency of $N+KL$ clock cycles; L is a tuning design parameter for speed of operation.

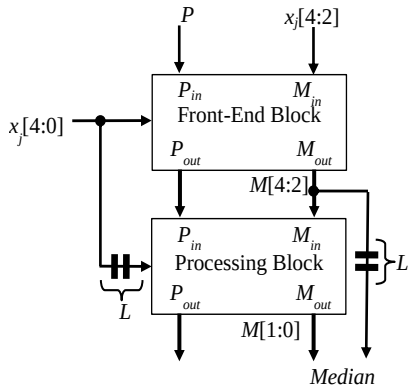

Fig. 2. Architecture to compute the median bits as in Table 1 using a front-end block of B = 3 bits followed by a processing block with B = 2 bits.

V. Timing Analysis and Implementation

A. Timing Analysis

The critical path delay T of Fig. 2 is essentially due to the APC accumulators of $l = \log_2 N$ bits each (of Fig. 1), and as the rightmost APC has $r = 2^B$ 1-bit inputs, then $T = \lfloor \log_2 2^B \rfloor + \log_2 N$ thus $T = B + \log_2 N$. A processing block in our previous method [6] had a critical path complexity of $3\log_2 N + 6$ for $B = 2$, so the processing block of Fig. 1 is three times faster than our previous method [6] for any $B < 6$. The critical path of the work in [3], $T_{[3]}$, is the delay cost of the Carry-Save Adder tree (CSA) and is at least of $\log_{1.5}(N/2) + \log_2 N$ to account for the final adder [13]. It follows that for $B \leq \log_{1.5} N/2$ a pipeline path here would be faster than a pipeline of the work in [3]. This is satisfied even for small values of N such as $N = 3$ and $B = 1$, which implies that the circuit here is faster for all practical cases values of N with a suitable choice of B. Interestingly, for $B = 1$, the work here is expected to be faster than the work in [3] for any N which suggests that the architecture in [3] may adopt the concept of APCs for a hybrid architecture. For $B > 1$ this work computes the median in W/B processing blocks while the work in [3] needs W processing blocks. It seems convenient to maintain B as small as 2, 3 or 4. This favors the parallel decoding and sign extension as shown in Fig. 1. Remarkably, the final accumulator in this work is still a ripple-carry adder. For the recent hardware sorting-based method in [4] the critical path goes through a chain of N-1 logic OR gates and therefore longer than the critical path of Figs. 1, 2. These latest sorting methods have been proposed for area-efficiency [4] or power [5].

B. Hardware Implementation

In order to verify the architecture presented here designs were expressed in RTL Verilog HDL and functionally verified by simulation. From the RTL design of the processing blocks using APC registers (similar to Fig. 1), circuit area and frequency of operation synthesis results is reported in Table III using ASIC TSMC 0.25μm technology. For a quick comparison the results of the previous work in [6] are also included. Clearly, using APC registers improves the frequency of operation as expected from the timing analysis. The front-end block offers an extra advantage in area especially when input integers are of $W \leq 12$ bits.

From Table IV it is seen that area scales well with parameter value N. A front-end block makes it easier to produce designs for any parameter value B; however the table suggests it is preferable to keep parameter B to 2, 3 bits (or make parameter $L > 2$ to increase frequency). Getting a circuit for Fig. 2 is subjected to implementation details at the RTL level. We explored maintaining the accumulation A_i coherent by fusing the decoder and sign extension of Fig. 1 into an ad-hoc decoder (a look up table) such that the critical path in the accumulation process is kept bounded by $\log_2 N$. In this case, the critical path could move into the logic towards the bottom of Fig. 1. This is the purpose of introducing the delay elements down the pipeline in Fig. 1. Table IV uses $L = 2$; this sort of tuning is best to be evaluated under the specific technology used to target the architecture and so it is not discussed in full detail in this paper. Observe the latency $N+KL$ paid by the architecture here is related to the number of blocks K ($K = W/B$ when each block is of B-bit slices); K remains small even in common practical cases ($W \leq 16$). The work in [3] has a latency of W but requires all samples in parallel and so needs WN input wires while this work needs only W input wires for streaming pipelined operation. The work in [4, 5] needs N processing blocks and so there is the tradeoff of evaluating overall

area, frequency of operation, and latency for a specific application. Note, Tables III and IV do not report results for a complete median architecture.

C. Extensions

The extensions to handle signed integers and the case of rank filtering as discussed in [6] remain valid. For keeping the paper self-contained we briefly mentioned them here. In order to handle signed integers, count the number of negative and positive integers within a window as C_0 and C_1 respectively so that $N = 2k+1 = C_0 + C_1$. Set the median position P to the first block of the computation to $P = k + 1 - C_1$ if $C_0 > C_1$ or to $P = k + 1 - C_0$ otherwise. The method remains unmodified if applied to the remaining W-1 bits of the input data set within the window. An order R filter for a set of N data elements has R data elements less or equal to the output [2]. The median is a rank filter with $R = k$, so this method to calculate the median behaves as a rank filter by setting initial median position to the first block of computation to $P = R + 1$ when accumulations A_i proceed right to left as performed here.

TABLE III
AREA AND FREQUENCY FOR MEDIAN ARCHITECTURE BLOCKS

W	Front-End Block		Processing Block		Old Block [6]	
	Gates	MHz	Gates	MHz	Gates	MHz
6	642	387	944	353	1169	166
8	801	355	991	353	1368	166
12	1010	324	1082	353	1768	166
16	1255	318	1173	353	2168	166

Front-End Block (Fig. 1, $B = 2$, $L = 2$), the previous block in [6] redesigned with APC accumulators (Processing Block, $B = 2$, $L = 2$), and the block of previous work in [6] ($B = 2$) for $N = 9$ and $W = 6$, 8, 12, and 16 bits.

TABLE IV
AREA AND FREQUENCY FOR A MEDIAN PARAMETERIZED BLOCK

W, N	8, 9	8, 25	8, 49	16, 9	16, 25	16, 49
B	A/f_{MHz}	A/f_{MHz}	A/f_{MHz}	A/f_{MHz}	A/f_{MHz}	A/f_{MHz}
2	801/355	1556/353	2761/337	1255/318	2584/268	4804/261
3	1172/330	2178/315	3314/319	1588/279	3083/258	5677/257
4	2065/309	3242/273	4703/266	2500/267	4319/248	7526/246

Front-end block Area (A) in gates, and frequency (f_{MHz}) in MHz. $L = 2$.

VI. CONCLUSION

Fundamental in median filtering methods for noise reduction in high quality imaging, the method for calculating the median given here makes faster decisions than previous hardware algorithms in the literature. The computation within each processing block is executed faster than before for any size of blocking bits (design parameter B). The median on a set of N integers completes after K (typically W/B) processing blocks for a serial pipelined stream of W-bit integers with a latency of $N+KL$, with L a tuning pipeline parameter for speed. It is also shown that this result holds irrespective of the actual values of parameter N or any combination of B and N. The use of full accumulative parallel counters circuitry is required for extending calculating the median in a parallel approach of accepting more than one integer at a time in streaming operation. The method is generic following a systematic number of steps from where different architectures and implementations can be derived. The method is also easily extended to be implemented as a fast programmed solution.

REFERENCES

[1] S. G. Akl, *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 39–58.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms.* Cambridge, Mass., MIT Press, 2003.

[3] D. Prokin and M. Prokin, "Low hardware complexity pipelined rank filter," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 57, no. 6, pp. 446–450, June 2010.

[4] R. D. Chen, P. Y. Chen, and C. H. Yeh, "Design of an area-efficient one-dimensional median filter," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 60, no. 10, pp. 662–666, Oct. 2013.

[5] R. D. Chen, P. Y. Chen, and C. H. Yeh, "A low-power architecture for the design of a one-dimensional median filter," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, to appear, 2015.

[6] J. Cadenas, G. M. Megson, R. S. Sherratt, and P. Huerta, "Fast median calculation method," *Electron. Lett.*, vol. 48, no. 10, pp. 558-560, May 2012.

[7] B. Parhami and C. H. Yeh, "Accumulative parallel counters," in *Proc. 23rd Asilomar Conf. on Signals, Systems, and Computers*, 1995, pp. 513–516.

[8] M. A. Neilsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2000.

[9] J. F. Wakerly, *Digital Design: Principles and Practices*. 4th Ed., Upper Saddle River, NJ: Prentice-Hall, 2006, Ch. 6.

[10] Q. Gan, J. M. P. Langlois, and Y. Savaria, "Parallel array histogram architecture for embedded implementations," *Electron. Lett.*, vol. 49, no. 2, pp. 99-101, Jan. 2013.

[11] Hieu B. V., Beak S. Choi S, Seon J. and Jeong T. T.: 'Thermometer-to-binary encoder with bubble error correction (BEC) for flash analog-to-digital converters (FADC)', in *Proc. 3rd Int. Conf. on Communications and Electronics*, 2010, pp. 102-106.

[12] S. Perreault and P. Hebert, "Median filtering in constant time," *IEEE Trans. Image Process.*, vol. 16, no. 9, pp. 2389-2394, Sept. 2007.

[13] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Designs*. New York, NY: Oxford University Press, 2000, pp. 125-140.