Published in IET Computers & Digital Techniques
Received on 18th September 2008
Revised on 22nd January 2009
doi: 10.1049/iet-cdt.2008.0119

ISSN 1751-8601

High-throughput one-dimensional median and weighted median filters on FPGA

S.A. Fahmy[1] *P.Y.K. Cheung*[2] *W. Luk*[3]
[1]*CTVR, Trinity College Dublin, Dublin 2, Ireland*
[2]*Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, UK*
[3]*Department of Computing, Imperial College London, London SW7 2AZ, UK*
E-mail: suhaib.fahmy@tcd.ie

Abstract: Most effort in designing median filters has focused on two-dimensional filters with small window sizes, used for image processing. However, recent work on novel image processing algorithms, such as the Trace transform, has highlighted the need for architectures that can compute the median and weighted median of large one-dimensional windows, to which the optimisations in the aforementioned architectures do not apply. A set of architectures for computing both the median and weighted median of large, flexibly sized windows through parallel cumulative histogram construction is presented. The architecture uses embedded memories to control the highly parallel bank of histogram nodes, and can implicitly determine window sizes for median and weighted median calculations. The architecture is shown to perform at 72 Msamples, and has been integrated within a Trace transform architecture.

1 Introduction

The median filter is a highly versatile non-linear filter that has been used extensively in a variety of domains. Its strength lies in its ability to filter out noise while minimally affecting the properties of the underlying signal. The median filter replaces a sample with the middle ranked value among all the samples within the sample window, centred around the sample in question. In this manner, it filters out samples that are not representative of their surroundings; in other words, outliers. In the image processing domain, a two-dimensional median filter allows for the removal of ‘salt-and-pepper’-type noise from an image without adversely affecting the underlying edges. The use of a linear filter (such as a Gaussian or mean filter) in this situation would cause a blurring of edges. The median filter can still degrade image quality somewhat, although the preservation of edges is paramount in the computer vision domain. Such filters within image processing are almost uniquely two-dimensional and have small window sizes.

Our recent work with the Trace transform [1, 2] highlighted the need for a hardware architecture to compute median and weighted median values on large one-dimensional windows. The Trace transform is a recently introduced algorithm that has been shown to perform well in a variety of image recognition and categorisation tasks, including image database search, face authentication and distortion correction [3]. It maps a standard image to an alternative domain and, while defining the spatial mapping, is general in terms of mathematical computation. The transform involves the computation of mathematical functions on lines crossing an image. Two of the functions typically used are the median and weighted median. Given that these lines can traverse the whole image, the number of sample points is of similar magnitude to the dimensions of the image, typically hundreds of pixels. Furthermore, given that the length of these lines is not fixed, a hardware architecture must cope with variable window sizes. The weighted median presents its own challenge in implementation terms, and we believe the work presented here and originally introduced in [4] to be the first hardware implementation of weighted median filters on large windows.

The median of a set of samples is often computed by sorting the input samples and then selecting the middle value. The weighted median can be computed in multiple stages: first expanding the weighted sample sequence, then sorting and finally locating the median. However, these methods are not suitable for incorporation within a streaming architecture because of the multi-stage approach.

We present a high-throughput hardware architecture that can compute the median and weighted median over a exibly sized window. The architecture is fully pipelined and thus implementable within the streaming architecture of the Trace transform presented in [10]. Indeed, the high performance afforded by this architecture was a main factor in achieving real-time (256×256 pixel 30 fps) performance for the Trace transform implementation described in that paper. This paper extends the work in [4], by generalising the architecture in terms of wordlengths of the bins and weights, including a detailed look at the weighted median implementation, allowing for fixed or variable median indices as well as a thorough investigation of field-programmable gate array (FPGA) resource utilisation for a wide range of different configurations.

The contributions of this paper are as follows:

- A highly parallel architecture for computing the median of input samples on large windows, which takes advantage of FPGA-embedded memories for parallel control (Section 4).
- An extension of the above architecture to computation of weighted medians (Section 5).
- A thorough investigation of resource requirements for different configurations (Section 6).

2 Definition

Given an input sequence $x_1, x_2, x_3, \ldots$, a window of size $2K+1$ is defined, centred on the ith value, x_i, as $W_i = \{x_{i-K}, x_{i-K+1}, \ldots, x_i, \ldots, x_{i+K-1}, x_{i+K}\}$. The output of the median filter, y_i, is thus the median of W_i; the middle value in the sorted list.

The weighted median is an extension of the standard median, wherein each sample in the window has an associated weight that determines how much that sample contributes to the final result. Weights can have fractional or integer values. From a computational perspective, this makes no difference as long as fractional weights are fixed point and lie within the same limits for all samples. Negative weights are undefined.

The input sequence for a weighted median filter can be written as

$$(x_1, w_1), (x_2, w_2), (x_3, w_3), \ldots$$

where x_i are the input samples and w_i are the corresponding weights. An integer weight would simply correspond to having w_i copies of sample x_i taken into account in the median calculation. Consider the example sequence

$$(1, 3), (5, 1), (2, 5), (4, 2), (7, 2), (3, 5), (4, 1)$$

After expanding, this becomes

$$1, 1, 1, 5, 2, 2, 2, 2, 2, 4, 4, 7, 7, 3, 3, 3, 3, 3, 4$$

To determine the median, this sequence must be sorted as follows

$$1, 1, 1, 2, 2, 2, 2, 2, 3, \mathbf{3}, 3, 3, 3, 4, 4, 4, 5, 7, 7$$

Finally, the middle value in the sequence, 3, is selected as the weighted median of this series.

It is important to note that for the weighted median, the size of the window is the sum of weights rather than the number of tuples received. So for the above sequence it is 19 and not 7. The median index can be calculated by halving this number and adding one, so in this case the median index is 10.

3 Related work

Much of the literature dealing with median filters in image processing is focused on two-dimensional filters of small size [5]. We consider these below, noting that the optimisations applied in these cases cannot map to the large-windowed one-dimensional requirements we are dealing with. Considering the simplicity of small two-dimensional median filters, work in the area has been generally saturated for a number of years, though developments at higher levels continue; an example is two-dimensional adaptive median filters [6]. Weighted median does not enjoy widespread use in image processing at present and thus little work has been done on efficient implementations.

Median filters have been implemented in hardware in a variety of ways. Reference [5] provides a very good review of the area. There are two main methods. The first is to maintain the input sample list in its original order and then pass it through some type of sorting network. The median value is then extracted from the relevant position in the ordered list. The other method involves sorting the samples as they enter the system. Of the first approach, the simplest implementation is the bubble sorting grid, where a grid of dual input sorters each swap their inputs to propagate the higher valued samples upwards and the lower valued samples downwards (or vice versa). The median is simply the middle sample of the grid output. An example of this architecture is shown in Fig. 1. This method is regular, yet its hardware requirements increase in proportion to the square of the window size and hence it is not scalable to larger windows. For a window of size $2K+1$, $(3K^2+3K)/2$ dual input sorters and $K+1$ registers are required as can be seen in Fig. 1.

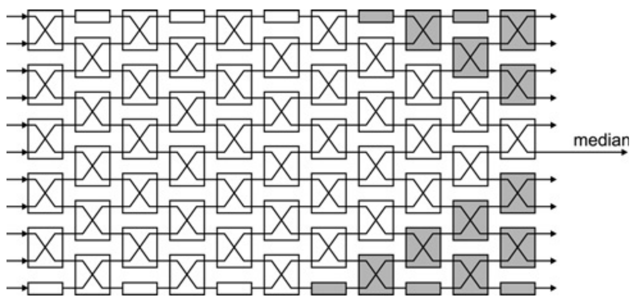

Figure 1 *A simple 11-sample bubble-sorting circuit layout*

The large blocks are compare- swap units that swap their inputs if necessary to propagate the larger values upwards and the smaller ones downwards: The small blocks are registers. Note that the shaded blocks are not required for median calculation

For small windows, simplifications can be made [7], where the columns and then rows of a two-dimensional window are each sorted using a triple-input sorter. Then only one diagonal needs to be sorted to give the median. This saves on hardware requirements. Karaman *et al.* [8] propose a change to the standard sorting network by dealing with samples in a bitwise manner, needing only single-bit sorters; however, their implementation is still proportional to the square of window size in area terms. The strength of regular array architectures is that they can be pipelined down to a single compare-swap stage. This results in high throughput. Benkrid and Crookes [9] create a sorting structure based on Quick Sort using a bit-voter block; the area requirements are $O(N)$. However, the scheme is only practical for small window sizes. Other methods that use fewer building blocks of higher complexity are described in [10–12]. Another method is that of threshold decomposition, as used in [13]; however, the architecture proposed relies on the window being of size 3×3 and uses three input adders and thus is not scalable to large windows. Systolic median architectures based on insertion sort have also been proposed [14]; in this case, the amount of hardware is proportional to the window size. Similarly for the implementation in [15]. In [16], the authors take advantage of the wide data buses on a development board to allow the median calculations for multiple pixels in parallel. The overlapping data between 3×3 windows are re-used and the sorting circuit is modified to reduce the number of compare-swap blocks. The proposed architecture is, however, limited to two-dimensional windows of 3×3 pixels and larger windows would not scale due to the sorting circuitry.

Another method for computation of the median of a sequence of numbers involves computing the cumulative histogram for this sequence, and then finding the index of the first bin total to exceed the median index. The principle is well established and known, having been mentioned in basic textbooks on image processing. Both [17] and [18] deal with software implementations of this algorithm running on general-purpose processors. The architecture first proposed in [4], and developed here, is the first hardware implementation of this algorithm. The high degree of parallelism that can be exploited in hardware, coupled with the independence of resource usage with regard to window size, is what makes this method so attractive as compared with a sorting structure. Furthermore, this method is elegantly extensible to the weighted median as will be shown.

4 Proposed architecture

4.1 General overview

The proposed architecture works by constructing a cumulative histogram of the input data. Each bin in this histogram represents one of the possible input values. The aim is that a new sample is accepted at every clock cycle; hence the whole histogram must update in a single cycle. The count values for each bin can then be compared with the median index, in parallel; the median value is the index of the first bin whose count exceeds the median index. This scheme has no sequential processing mode, and hence can continue to give median values in real time as new data arrive. In this section, we build the architecture starting with a basic fixed window median architecture. We then show how this can be extended to sliding windows, by keeping track of old samples that fall out of the window.

Since the application domain in this case is video processing, 8-bit unsigned numbers (let $l = 8$) have been assumed. This means there are $2^8 = 256$ possible input values, and so a rank of 256 bins is used to store the cumulative histogram. When an input value is received, the bin corresponding to the sample value is incremented. For a cumulative histogram, each subsequent bin must also be incremented. In software, this is normally done as an additional step after the histogram has been fully populated. A pass through all the bins adds the value of the previous bin to each bin. Hence, the value stored in the final bin will always be equal to the number of input samples received. The median is then simply the first bin whose count reaches or exceeds the median index.

For example, if the median is to be calculated over a window of 101 elements, that is, $2K+1 = 101$, $K = 50$, then the 51st or generally $(K+1)$th element in the ordered list is required. Using the histogram, find the first bin whose count is 51 or above; this gives the median of the input samples, since the 51st ordered element must lie in this bin.

To implement this in hardware, a rank of parallel bin nodes is instantiated. A separate register is required to keep a count for each of the possible input values. Fig. 2 shows the design for a histogram node. For all the registers to be updated in parallel, each register also requires its own incrementer, which is activated only when that bin needs to be incremented. (Recall that to construct a histogram, only the bin with an index corresponding to the input sample

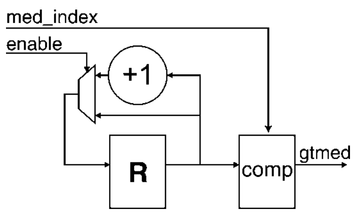

Figure 2 *A histogram bin node processor*

needs to be incremented.) Hence, each bin has an enable input (enable in Fig. 2) that determines whether it should be incremented in the current clock cycle, and a median index input (med_index). The output is a single binary value (gt_med) that is 1 when the value equals or exceeds the median index and 0 otherwise. 256, or in the general case of l-bit samples, 2^l, such nodes are required in the proposed system. Note that the width of the bin registers depends on the window size required. This will be investigated further in Section 6.

A circuit composed of such processors would yield the histogram of the input signals. In order to compute the cumulative histogram, some further processing is needed. As mentioned above, it is possible to separate the construction of the cumulative histogram and do this as a subsequent step. This, however, would be wasteful, as the accumulation for each bin would have to be done in turn, taking 256 cycles in total. One possible alternative approach is to instantiate a comparator for each bin, and compare the input sample value to the index of each bin. Those bins with an index greater than or equal to the input sample value would be incremented. However, this would be costly in terms of hardware, since each bin would require its own l-bit comparator.

Another approach would be to connect each bin to the previous one, such that if the previous bin is being incremented, then it increments too. However, this would slow down the system down significantly, since that incrementation signal would need to propagate through 256 stages in the worst case, all in one clock cycle. Analogous to this is the carry chain in a carry-ripple adder.

A more efficient method, which takes advantage of the heterogeneous resources on modern FPGAs, is to use embedded Block RAMs on the FPGA as a ROM to store the bin access patterns. For the 8-bit inputs previously mentioned, a 256×256-bit ROM would be required to decode the 8-bit number into a 256-bit signal, where each bit represents the select input shown in Fig. 2, to the corresponding bin; each bit of the output addresses a single-bin node processor. The access patterns stored in the ROM ensure that the correct bins are enabled for any given input sample. The contents of the ROM are shown in Table 1, whereas an overview of the circuit is shown in Fig. 3.

This method of constructing a cumulative histogram is highly efficient and allows for a fully updated histogram in every cycle. This method has also subsequently been adapted for histogram equalisation on images [19] and shown to perform significantly better than a software implementation on a Graphics Processing Unit (GPU) [20]. Note that histogram generation is just one part of the median and weighted median implementation.

Table 1 Access pattern ROM contents

Address	Contents[0:255]
0	0xFFFFFFF... FFFFF
1	0x7FFFFFF... FFFFF
2	0x3FFFFFF... FFFFF
3	0x1FFFFFF... FFFFF
4	0x0FFFFFF... FFFFF
5	0x07FFFFF... FFFFF
⋮	⋮
253	0x0000000... 00007
254	0x0000000... 00003
255	0x0000000... 00001

Now, the count value for each bin is compared with the median index (in this example, 51), resulting in a 0 if the bin count is smaller, and a 1 if it is equal or larger. Hence the result for all bins before the one containing the median will be 0, and all the others will be 1. A priority encoder can then be used to find the index of the first bin in the series of 1's. A priority encoder takes a B-bit input in which there are b zeros followed by $B - b$ ones, and returns $b + 1$, the index of the first 1. This gives the median of the input.

4.2 Sliding window implementation

Thus far, the system takes a sequence of samples and returns the median for a fixed window, once it has been filled. Such a circuit, however, is not useful, since

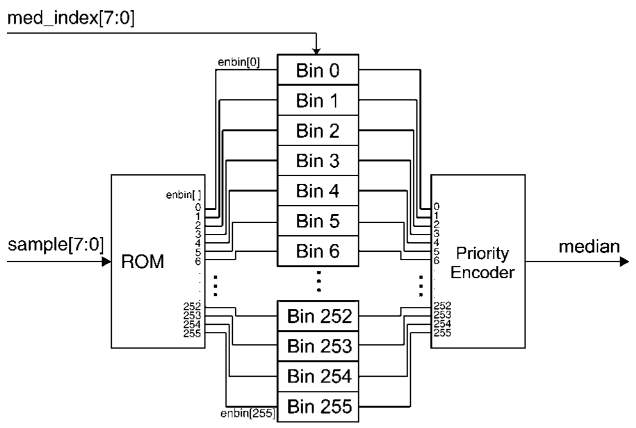

Figure 3 *Histogram-based median filter architecture*

normally the median must be computed for a sliding window. This means that in each cycle, the window moves one sample down the sequence, discarding the oldest sample and adding the newest into the window. To implement this algorithm for sliding windows, some changes must be made. Consider that now while constructing a histogram, with each new sample that enters, the oldest sample is removed from the window, and thus its effect on the histogram must also be negated. This, however, only happens after the window has become full. Hence some way of keeping track of the old samples, knowing when the window has become full for the first time, and some way of updating the histogram based on the new and oldest samples must be devised.

Firstly, a FIFO buffer is used to store the samples for the window over which the median must be found. When a new sample is received and the window is full, the oldest sample is removed from the FIFO (oldsamp in Fig. 6). Updating the histogram requires all bins corresponding to the access pattern for the oldest sample to be decremented. At the same time, the bins corresponding to the new input sample must be incremented. This can all be done in one cycle, by simply leaving any bins that are included in both sets unchanged, since they increment and decrement at the same time. Bins that are only enabled by the access pattern of the new sample are incremented, whereas bins enabled only by the access pattern of the removed sample are decremented. Updating the histogram in this fashion means that it is up-to-date in every clock cycle, and there need not be a pause in the input samples. The new node design is shown in Fig. 4.

On-chip Block RAMs are particularly useful for this architecture. Because these RAMs are dual ported on the target architecture, it is possible to extract the enable signals for both the new and oldest samples from the access pattern ROM in parallel. These can then be processed to determine which bins are incremented and decremented, as illustrated in Fig. 5.

To implement this, a simple two-input, two-output look-up table is required to determine the resultant action. This is shown in Table 2. This small logic function must be

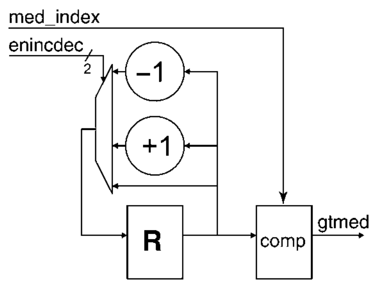

Figure 4 *A bin node for the sliding window implementation*

Enincdec is simply a concatenation of the two bits from the ROM lookup

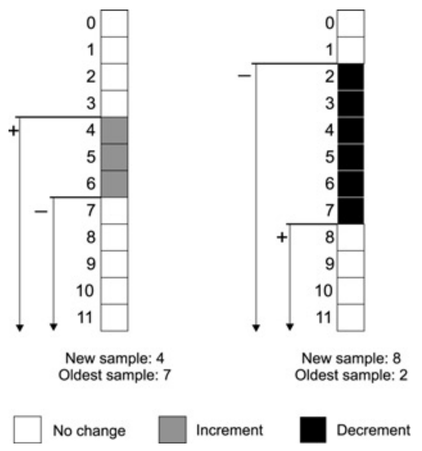

Figure 5 *Application to sliding windows*

The arrows aside the bins show the access patterns for the oldest (−) and new (+) samples. The leftmost example shows a new sample value of 4 arriving whereas the oldest sample is of value 7. Only bins 4–7 need to be incremented; all others retain their current values. The rightmost example shows a new sample of value 8 arriving, whereas the oldest sample is of value 2. Only bins 2–8 need to be decremented; others are left alone

implemented for each bin, and this represents the enincdec input shown in Fig. 4. Recall that as the window is filling with values the first time, no subtractions take place, since this would mean that the histogram would never fill up with values. As such, a single valid bit is appended to each input sample. This propagates through the FIFO and emerges at the final stage of the FIFO (windowfull) only when a full window of values has been received. This bit gates the subtraction control signal, so no subtraction can take place until it emerges. The revised architecture is shown in Fig. 6.

5 Weighted median architecture

The architecture thus far computes a standard median, on a fixed window size. To implement weighted median, further changes to the architecture in Fig. 6 are needed. Recall that

Table 2 Extra sliding window logic

OldEn	NewEn	enincdec[1:0]
0	0	00
0	1	10
1	0	01
1	1	00

The signals OldEn and NewEn are the enable signals for the bin resulting from the ROM lookup of the oldest and new samples, respectively. *Enincdec* is the signal that instructs the bin counter whether to increment (10), decrement (01) or do nothing (00)

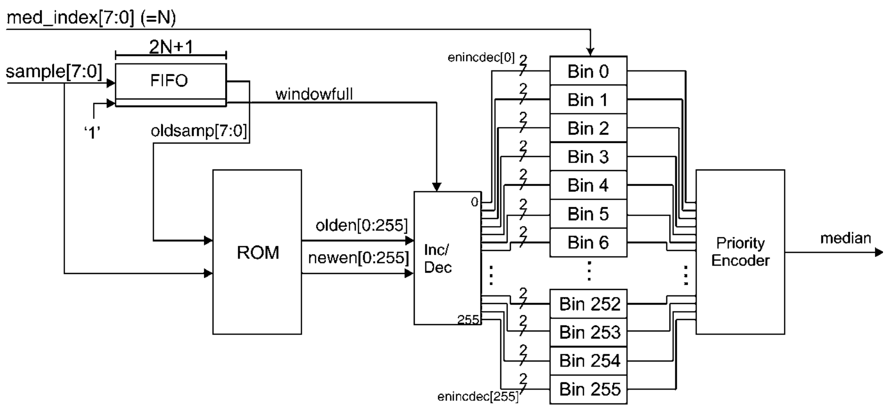

Figure 6 *Architecture of the sliding window median filter*

the weighted median is computed on samples that have associated weights and that those weights are equivalent to duplicating the sample the corresponding number of times. Further recall that the window size, and thus the median index, is dependent on these weights. To construct the histogram for weighted samples, rather than increment each bin for corresponding samples, the weight of that sample is added to the corresponding bin. The cumulative histogram is constructed as per the standard median.

To make the necessary changes in hardware, another input signal is introduced to provide the weights. Instead of a simple incrementer, each bin processor must now add the weight value. Just as with the standard median, it is possible to keep the histogram fully updated at each clock cycle. Another FIFO is instantiated, to keep track of the old weights that fall out of the window (oldweight in Fig. 7). For bins enabled by the access pattern for the oldest sample falling outside the window, the weight of that sample (oldweight) is subtracted. For those bins enabled by both access patterns, the difference of the two weights (weightdiff) is added (while being careful to maintain the correct sign). For those bins enabled only by the access pattern for the new sample, the new sample's corresponding weight (weight) is added. The resultant architecture is shown in Fig. 7. The three signals fed into each of the bins are the weight of the new input sample, the difference in weights and the weight corresponding to the sample falling outside the window.

The rank, or position, of the median is not known in advance for weighted median. Consider the expansion of the sequence shown in Section 1, and it becomes clear that the number of 'real' samples received is equal to the sum of sample weights. Hence, the index of the median must be half of that plus one, which is simply a right shift and increment in hardware. In the proposed architecture, the difference of the two weights (that of the new sample and that of the oldest, weightdiff in Figure 7) is simply added to a register on each clock cycle. This maintains the current

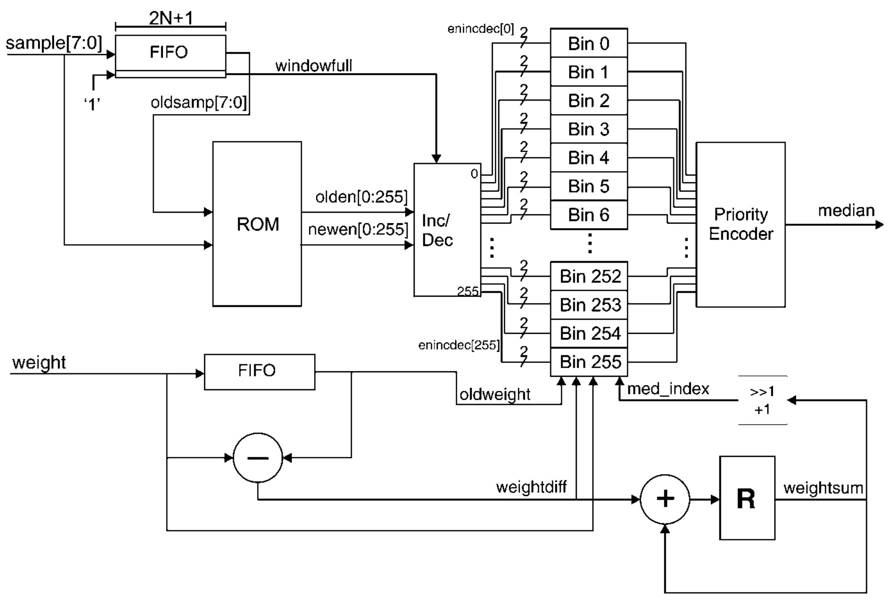

Figure 7 *Architecture of the weighted median filter*

sum of weights (weightsum). This is right shifted to divide by two and incremented and fed into each of the bins as med_index, where it is used for the comparison.

The wordlength of each bin register must be wide enough to accommodate the maximum sum of the weights to prevent overow. In order to do this, the width of the bin counters must be equal to $\log_2$ of the window size plus the width of the weights.

This weighted median implementation also opens the door to a flexible standard median implementation. Rather than have a fixed median index, it is possible to use the count of input samples received to determine the median index. This allows us to use a single implementation for multiple window sizes. This is a requirement for implementation within the Trace transform.

6 Implementation results

Implementation of the above designs was originally coded in Handel-C and compiled using the Celoxica DK Compiler. The target device in this case is a Xilinx Virtex II 6000, as found on the Celoxica RC300 development board. For comparison, an alternative implementation of the median filter based on the sorting grid mentioned in Section 3 was also synthesised. Note that this comparison is for illustrative purposes only. The key motivator for this architecture is the fact that it is completely flexible with regard to window size, even for subsequently processed windows. Other architectures are designed as point solutions for a single window size. Sorting architectures also fail to deal elegantly with weighted median calculation.

Using Handel-C was found to give acceptable area and speed results for the sorting-grid architecture. However, the proposed circuit was found to have a high clock period. For the proposed architecture, the area usage was halved and the clock period reduced by over 60% when it was re-implemented in VHDL. The reason for this disparity is a function of how Handel-C is implemented in hardware. A Handel-C circuit functions using token passing, effectively enabling subsequent parts of the circuit. The problem arises when there are a large number of units that have to be enabled in parallel, as with the 256 bins in this case. The single token passing signal must be fanned out to a huge number of circuit elements and this fan-out introduces significant routing delay.

6.1 Design variations

In order to thoroughly investigate the proposed architecture, a number of variations were considered. Fixed window implementations were ignored, since they are of little use, returning a single result for a whole window. Instead, sliding window implementations were favoured because of their computation of a new result every cycle. A number of design parameters were varied, leading to multiple implementations. Before discussing these, consider the parameters that might affect area. Firstly, all implementations were synthesised for sample widths of 8-bits. This is an assumption that is valid for most of the calculations one would wish to conduct on images. Furthermore, this is the only significant limiting factor for this design because the number of bins varies exponentially with the sample wordlength.

Clearly, changing the wordlength of input samples would also impact the number of memories required for the design. The 8-bit samples require a 256×256-bit memory, whereas the 10-bit samples require a 1024×1024-bit memory. This would require a re-implementation of the design, although this can be done through a simple parameter change and re-synthesis. If complete accuracy is not necessary, it is also possible to simply use 256 bins for 10-bit samples by slicing off the upper 8-bits for the median calculation. Sample wordlength represents the main scaling dependency for this architecture. Since this implementation is to be used in the Trace transform, 8-bit wordlengths are sufficient.

The counters in each of the bins need to be wide enough to accommodate the maximum count, equal to the maximum number of samples to be considered, which is equivalent to the window size. Hence the width of the bins is equal to the base-2 logarithm of the window size. One can set this arbitrarily to a fixed number such as 8-bits as we previously did in [4]. This would allow for window sizes up to 255 samples. However, to keep the design as compact as possible it should be set to the appropriate width. The window size also affects the length of the FIFO buffer used to track older samples. This buffer is equal in length to the window size. Finally, one may choose to implement a design that uses a fixed window size, or one in which the window size is determined by the number of samples entering the system. The advantage of the second method is that the window size can be changed at runtime. The first method would synthesise a fixed value comparator; although this saves area, it is less flexible.

6.2 Synthesis results

All designs were synthesised to run at 72 MHz. All designs used 8 Block RAMs to implement the bin selection lookup. Each on-chip 18 Kb Block RAM can be configured in a number of width and depth configurations. The shallowest configuration is 512×36 bits. Hence a 256×256 memory would require eight of these side by side.

The first set of results, shown in Fig. 8, shows the area usage for basic sliding window implementations and how this varies with the window size for each of three metrics: Look-Up Tables (LUTs), Flip-Flops (FFs) and Slices. Note that slices are the realistic area metric, but FFs and LUTs are shown as they map well to discussion of the architecture. These implementations were for fixed window

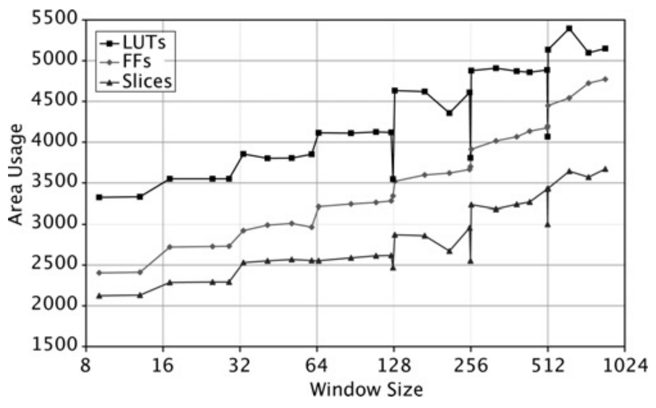

Figure 8 *Graph of synthesis results for various window sizes*

sizes using hard-wired fixed value comparators. The vertical lines in the graph indicate the boundaries of different wordlengths for the bin counters.

It is clear from the graph that each time the counter wordlength requirements increase by one bit, there is a distinct jump in area requirements, in the general case. FF usage has a general rising trend, even between designs with counters of the same wordlength. This is due to the increasing size of the FIFO buffer; this also explains the increasing gradient of the FF segments since the window size is on a logarithmic scale.

The considerable variations in LUT usage can be put down to the optimisation of the fixed value comparator. When comparing values to a fixed number, and depending on the value of the fixed number, not all bits need to be taken into account. The synthesis tools will optimise the comparators as required. This is most evident for window sizes of 127, 255 and 511 in the graph. The binary representation of these values is 1111111, 11111111 and 111111111, respectively. The median index will thus be half plus one, giving 1000000, 10000000 and 100000000, respectively. When comparing a number to determine whether it is greater than or equal to these numbers, only a single bit needs to be tested. This means that the comparator is reduced to a one-bit comparator, resulting in a significant reduction in area. Other fluctuations are the result of similar optimisations applied by the tools.

The graph also shows a lack of a jump in the Slice count around the 64- and 512-sample window sizes. This can be attributed to the synthesis tools packing the LUTs and FFs differently, resulting in a more dense arrangement within the slices. Again, the designer has little control over this.

The general trend for area requirements can thus be described as being of the form $A + \log_2 B$, where B is the window size and A is the fixed area required by the rest of the design regardless of window size. The graph in Fig. 9 shows how this compares very favourably with the area usage of the standard sorting grid architecture that was also implemented. The sorting grid architecture's area requirement increases exponentially with regard to window size. The point at which the proposed architecture becomes more efficient is at a window size of approximately 23 samples. Note that other sorting algorithms can be used. However, the best case complexity for a window of N samples is of order $N\log(N)$, so the proposed algorithm remains advantageous, especially for large window sizes.

The next variation involves generalised comparators. In these implementations, the median index is computed automatically from the value of the counter in the last bin. Recall that the last bin in a cumulative histogram contains the count of the total number of samples in the system. This can be halved and incremented to give the median index. The strength of this system is that it allows for variable window sizes. Clearly, the area requirements will increase, since the comparators cannot now be optimised by the synthesis tools and must be full l-bit comparators, where l is the wordlength of the bin counter. The graph in Fig. 10 shows how each of the area metrics increases when this modification is made. A window size was selected from the middle of the range of values used for the first set of results and an equivalent circuit was implemented but with a variable median index. The window sizes used for each of the different wordlengths were 13, 25, 51, 109, 211, 387 and 739, respectively. There was no need to synthesise the full range of window sizes, as the only difference would be in the FIFO length. The dotted lines in the graph indicate the requirements for the fixed comparator equivalents. The number of FFs remains almost constant since the FIFO is not affected by this architectural change. The LUT usage, however, increases by between 23 and 26%, whereas the slice count increases by between 19 and 22%. An implementation with generalised comparators is the one used for the Trace transform.

The final variation of designs was the weighted median implementation. Recall that each sample in this implementation has an associated weight; this weight is

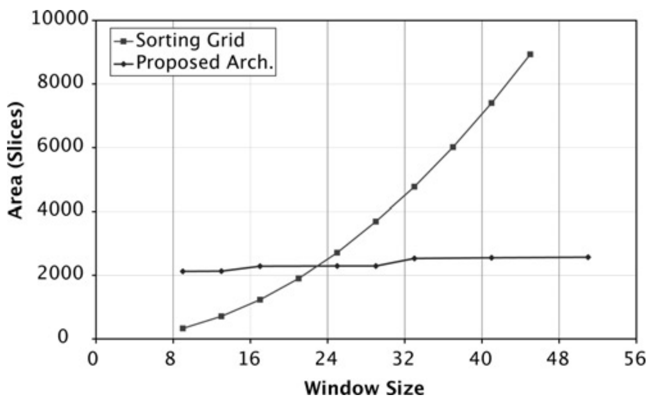

Figure 9 *Comparison of area requirements for proposed algorithm and sorting grid*

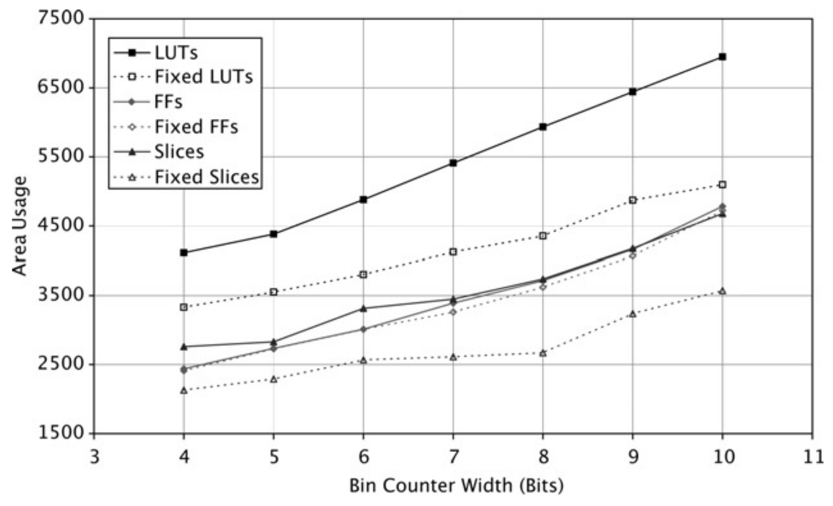

Figure 10 *Comparison of area requirements for fixed value and variable comparators*

used to update the histogram. This introduces another variable. the width of this weight in bits. Clearly, this will have an effect on the window sizes that can be implemented for each bin size. If the bin width is set to p-bits, then for the standard median it can accommodate a window size of up to $2^p - 1$ samples. For the weighted median, this window size will depend upon the width of the weights. If the weights are given widths of q-bits, then the maximum window size for a bin width of p-bits is $2^{p-q} - 1$. Hence, increasing the width of the weights means wider bins are required for an equivalent window size.

The results of this set of implementations are shown in Fig. 11. It can be seen that increasing the width of either the bin counter or weight has a similar effect. Furthermore, the area required for a weighted median implementation with weights 2-bits wide is not very different from the generalised version of the standard median filter. As the width of the weights increases, the resource requirements of the weighted median implementation begin to exceed the generalised median more significantly.

6.3 Discussion

Through developing a parameterised design, it is easy to tailor the implementation to specific requirements in terms of wordlengths and window size. The only assumption that holds for all the above designs is that the input samples are 8-bits wide, as one would find reasonable in the sphere of image and video processing. The extensibility of the original design coupled with full pipelining has meant that all these derivatives could be derived from one architecture, and all can run at 72 MHz, returning one result in every clock cycle. The throughput is thus 72 Msamples/s. Note that this cannot be converted to frames per second since the implementation is not designed as a spatial filter; it is an arithmetic unit. However, for illustration's sake, the

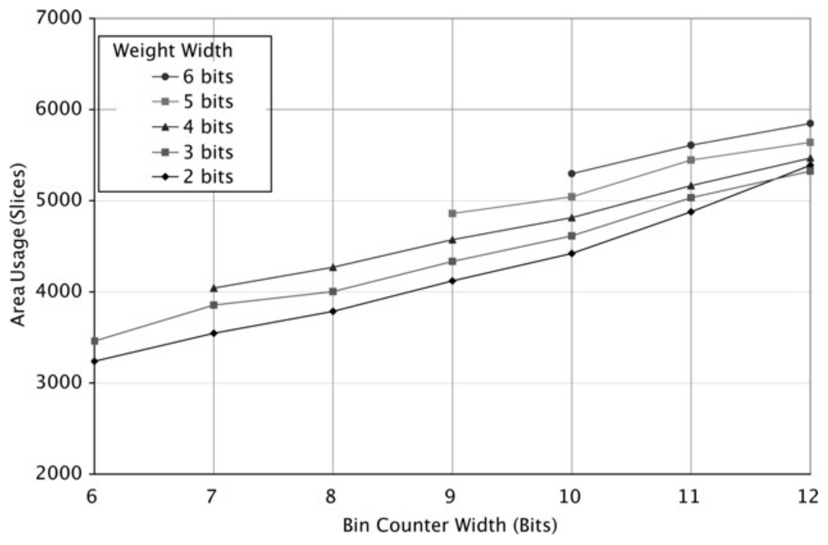

Figure 11 *Area requirements for various weight and bin width combinations*

computation throughput of this circuit would equal 234 frames per second for 640×480 pixel images.

Comparing with sorting-based methods is not straightforward. It is worth noting that optimal sorting topologies differ according to the size of the window to be sorted. Although an optimal design for a specific window size may be obtained, this may not be optimal when applied to different sizes. Furthermore, the topologies can be non-regular, which precludes their use in a generalised fashion for exible window sizes. This is one of the strengths of the proposed architecture since it is not dependent on window size. A further point to consider is that even if a sorting topology's area requirements may increase proportional to $N \log N$, the connectivity for more complicated sorting structures could present a challenge to FPGA routing tools as window sizes increase, resulting in higher clock periods and hence lower throughput as window sizes increase. This is a topic for further research.

The 72 MHz clock speed is achieved using the Xilinx tools with their standard settings. Routing delays due to the heavy routing required for a massively parallel design contribute to this. Furthermore, we are limited by the pipeline speed of the Block RAM primitives as well as the fact that we are using them stitched together as part of a larger logical memory. We expect the performance to scale well with improvements in the target architecture.

7 Conclusions and future work

This paper presented an alternative implementation of median filtering for arbitrarily large one-dimensional windows. The architecture is highly scalable in terms of window size. The design also allows for a exible window size that can change from one window to the next. The use of heterogeneous FPGA resources allow the circuitry to be simplified and fully pipelined. The area requirements were compared with that of a standard sorting-grid architecture and show the efficiency of this method for larger windows. An extension to weighted median calculation was also shown, which has a modest impact on resource requirements. A full analysis of area requirements for fixed-sized windows, flexible windows and the weighted median implementation was shown. The presented method is elegant in its flexibility with regard to window size. Of course, for very small windows other techniques may be more compact. However, for large windows, or systems where flexibility in window size is needed, or for weighted median calculation, the proposed method is scalable, offers a throughput of 72 Msamples/s and uses area equivalent to 15% of the target FPGA. The method could be extended to window sizes orders of magnitude larger if required. The only impact on the design is the need for wider bin counters in the histogram. A derivative of this architecture was incorporated into the Trace transform system described in [2].

We intend to investigate further uses of the cumulative histogram architecture within image and signal processing. We are also investigating further applications of the ROM-based parallel control scheme used to control histogram bins in this architecture.

8 Acknowledgment

This work was undertaken while the author S.A. Fahmy was a PhD student at Imperial College London.

9 References

[1] FAHMY S.A., BOUGANIS C.-S., CHEUNG P.Y.K., LUK W.: 'Efficient realtime fpga implementation of the trace transform'. Proc. Field Programmable Logic and its Applications, 2006, pp. 555–560

[2] FAHMY S.A., BOUGANIS C.-S., CHEUNG P.Y.K., LUK W.: 'Real-time hardware acceleration of the trace transform', *J. Real-Time Image Process.*, 2007, **2**, (4), pp. 235–248

[3] KADYROV A., PETROU M.: 'The Trace transform and its applications', *IEEE Trans. Pattern Anal. Mach. Intell.*, 2001, **23**, (8), pp. 811–828

[4] FAHMY S.A., CHEUNG P.Y.K., LUK W.: 'Novel FPGA-based implementation of median and weighted median filters for image processing'. Proc. Int. Conf. Field Programmable Logic and Applications, 2005, pp. 142–147

[5] RICHARDS D.S.: 'VLSI median filters', *IEEE Trans. Acoust., Speech Signal Process.*, 1990, **38**, (1), pp. 145–153

[6] VAÍAND Z., SEKANINA L.: 'Novel hardware implementation of adaptive median filters'. Proc. IEEE Workshop Design and Diagnostics of Electronic Circuits and Systems, 2008, pp. 110–115

[7] BATES G.L., NOOSHABADI S.: 'FPGA implementation of a median filter'. Proc. IEEE TENCON '97 IEEE Region 10 Ann. Conf., 1997, vol. 2, pp. 437–440

[8] KARAMAN M., ONURAL L., ATALAR A.: 'Design and implementation of a general-purpose median filter unit in CMOS VLSI', *IEEE J. Solid-State Circuits*, 1990, **25**, (2), pp. 505–513

[9] BENKRID K., CROOKES D., BENKRID A.: 'Design and implementation of a novel algorithm for general purpose median filtering on FPGAs'. Proc. 2002 IEEE Int. Symp. Circuits and Systems, 2002, vol. 4, pp. 425–428

[10] YU H.-S., LEE J.-Y., CHO J.-D.: 'A fast VLSI implementation of sorting algorithm for standard median filters'. Twelfth Annu. IEEE Int. ASIC/SOC Conf., 15–18 September 1999, pp. 387–390

[11] CHEN C.-T., CHEN L.-G., HSIAO J.-H.: 'VLSI implementation of a selective median filter', *IEEE Trans. Consum. Electron*, 1996, **42**, (1), pp. 33–42

[12] BREVEGLIERI L., PIURI V.: 'Digital median flters', *J. VLSI Signal Process. Syst. Signal, Image, Video Technol.*, 2002, **31**, (3), pp. 191–206

[13] BURIAN A., TAKALA J.: 'VLSI-efficient implementation of full adder-based median filter'. 2004 IEEE Int. Symp. Circuits and Systems, 23–26 May 2004, vol. 2, pp. 817–820

[14] GUO S., LUK W.: 'An integrated system for developing regular array designs', *J. Syst. Archit.*, 2001, **47**, (3–4), pp. 315–337

[15] RAVI TEJA V.V., RAY K.C., CHAKRABARTI I., DHAR A.S.: 'High throughput VLSI architecture for one dimensional median filter'. Proc. IEEE Int. Conf. Signal Processing, Communications and Networking, 2008, pp. 339–344

[16] VEGA-RODRÍGUEZ M.A., SÁNCHEZ-PÉREZ J.M., GÓMEZ-PULIDO J.A.: 'An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems'. Proc. 10th Mediterranean Conf. Control and Automation, 2002

[17] ANGELOPOULOS G., PITAS I.: 'A fast implementation of two-dimensional weighted median filters'. Proc. 12th Int. Conf. Pattern Recognition, 9–13 October 1994, vol. 3, pp. 140–142

[18] HAYAT L., FLEURY M., CLARK A.F.: 'Two-dimensional median filter algorithm for parallel reconfigurable computers', *IEE Proc. Vision, Image Signal Process.*, 1995, **142**, (6), pp. 345–350

[19] ALSUWAILEM A.M., ALSHEBEILI S.A.: 'A new approach for real-time histogram equalization using FPGA'. Proc. 2005 Int. Symp. Intelligent Signal Processing and Communication Systems. ISPACS, 2005, pp. 397–400

[20] COPE B., CHEUNG P.Y.K., LUK W.: 'Bridging the gap between FPGAs and multiprocessor architectures: a video processing perspective'. Int. Conf. Application-specific Systems, Architectures and Processors (ASAP), 2007, pp. 308–313

doi: 10.1049/iet-cdt.2008.0119