CHAPTER 7

MEDIAN FILTER AND EHW

Image Processing is an ever expanding and dynamic area with applications impacting our everyday life in such diverse fields as medicine, space exploration, surveillance, and authentication. In real-time image transmission, images are mostly affected by additive noise (like Gaussian noise) or impulse noise (e.g. salt and pepper noise). The received images should be enhanced in such a way that the Mean Square Error (MSE) of the received image with reference to the original image is minimized. The image enhancement requires a series of filtering operations on the received image. These filters are serially connected, since the image processing design is divided into a number of processing stages. Implementing such filters on a general purpose computer is straightforward, but not very time efficient, due to constraints on processor speed and available memory. The run-time efficiency of filters can be improved through the use of specially designed Application Specific Integrated Circuits (ASICs) and Digital Signal Processors (DSPs). Fast execution times for complex computations can be achieved by optimizing the circuits for the specific application (Gudise and Venayagamoorthy 2000).

System on Chip (SoC) platforms based on ASICs and DSPs can exploit the parallelism and pipelining algorithms to allow for greatly reduced execution times. However, these designs result in a fixed hardware architecture and circuitry, and therefore cannot implement complex systems in an efficient and flexible manner. These limitations can be overcome by
leveraging recent advances in Reconfigurable SoCs based on Field Programmable Gate Arrays (FPGAs). The latest version of these reconfigurable systems introduces the concept of ‘Dynamic Run-time Partial Reconfiguration’, where only a small portion of the circuitry is modified at run-time while the system remains functioning. Implementing image processing algorithms on reconfigurable hardware allows for the dynamic selection of the filter depending on the characteristics of the received image. This improves the image quality for a wide range of images, while simplifying the debugging and verification process. The median filter is often used to remove "shot" noise, pixel dropouts and other spurious features of a single pixel extent, while preserving the overall image quality (Goldstein et al 2000). In contrast, low pass filters would only blur the noise instead of removing it. An efficient algorithm to determine the median is desired, because this operation often has to be repeated millions of times for filtering large images.

One simple approach, which is often found in image processing textbooks, is to calculate the 3x3 median, using a simple sorting algorithm, like bubble sort or quicksort, and pick the 5th element after the sorting. An improvement on this simple technique is only to sort until the 5th element is determined. For example, a modified bubble sort can be used to sort until the 5th element. This approach yields 30 comparisons for one median calculation. A better approach is published in the first Volume of the Graphics Gems series by (Paeth 1990). This approach is based on a successive minmax-elimination: the minimum and the maximum of the first six elements are determined and eliminated. Then the 7th element is added to the remaining four of the first pass, and the minimum and the maximum of the five elements are determined and eliminated. This scheme is repeated until the 9th element is added to the remaining two and the minmax-elimination results in the median of all nine elements. This algorithm needs 20 comparisons per median.
The drawback, that the algorithm does not use spatial coherence, can easily be remedied: Simply calculate two neighboring medians in one step, where the first minmax-elimination is computed from the common six elements, and can be used for both medians. This improvement would result in a better performance using only 16.5 comparisons per median. A comparison of other median filtering algorithms can be found in (Juhola et al. 1991), but these techniques are not optimized for the common 3x3 kernel. The algorithm proposed here uses the coherence information between neighboring median calculations more efficiently, and therefore, needs only a maximum of 9.5 comparisons per median. The average number of comparisons is even smaller.

7.1 ALGORITHMIC CONCEPT

This research is focused on developing hardware implementations of image processing algorithms for use in an FPGA-based video processing system. This chapter discusses these algorithms and their software implementations in MATLAB.

7.2 WINDOWING OPERATORS

In image processing, several algorithms belong to a category called windowing operators. Windowing operators use a window, or the neighborhood of pixels, to calculate their output. For example, a windowing operator may perform an operation, like finding the average of all pixels in the neighborhood of a pixel. The pixel around which the window is found is called the origin. Figure 7.1 shows a 3 by 3 pixel window and the corresponding origin.
This project is based on the usage of image processing algorithms using these pixel windows to calculate their output. Although a pixel window may be of any size and shape, a square 3x3 size was chosen for this application, because it is large enough to work properly and small enough to implement efficiently on hardware.

7.3 RANK ORDER FILTER

The rank order filter is a particularly common algorithm in image processing systems. It is a nonlinear filter; so, while it is easy to develop, it is difficult to understand its properties. It offers several useful effects, such as smoothing and noise removal. The median filter, which is a rank order filter, is especially useful in noise removal.

7.4 ALGORITHM

This filter works by analyzing a neighborhood of pixels around an origin pixel, for every valid pixel in an image. Often, a 3x3 area, or window of pixels is used to calculate its output. For every pixel in an image, the window of the neighboring pixels is found. Then the pixel values are sorted in the ascending, or rank, order. Next, the pixel in the output image corresponding to the origin pixel in the input image is replaced with the value
specified by the filter order. The rank order filter can be represented by the following lines of the pseudo-code:

7.5 **PSEUDO-CODE FOR RANK ORDER FILTER**

order = 3 (this can be any number from 1 -> # pixels in the window)
for loop x -> number of rows
for loop y -> number of columns
window_vector = vector consisting of current window pixels
sorted_list = sort(window_vector)
output_image(x,y) = sorted_list(order)
end
end.

Figure 7.2 shows an example of this algorithm for a median filter (order 3), a filter that is quite useful in salt-and-pepper noise filtering. Since the rank order filter uses no arithmetic, a mathematical description is difficult to represent efficiently.

![Figure 7.2 Rank Order Filter Operation](image)
As is evident from the Figure 7.2, it is possible to use any order up to the number of pixels in the window. Therefore, a rank order filter using a 3x3 window has 9 possible orders, and a rank order filter using a 5x5 window has 25 possible orders. No matter what the window size used in a particular rank order filter is, using the middle value in the sorted list will always result in a median filter. Similarly, using the maximum and minimum values in the sorted list always results in the flat dilation and erosion of the image, respectively. These two operations are considered part of the morphological operations.

7.6 ADAPTIVE MEDIAN FILTER

The Adaptive Median Filter is designed to eliminate the problems faced with the standard median filter. The basic difference between the two filters is that, in the Adaptive Median Filter, the size of the window surrounding each pixel is variable. This variation depends on the median of the pixels in the present window. If the median value is an impulse, then the size of the window is expanded. Otherwise, further processing is done on the part of the image within the current window specifications. ‘Processing’ the image basically entails the following: The center pixel of the window is evaluated to verify whether it is an impulse or not. If it is an impulse, then the new value of that pixel in the filtered image will be the median value of the pixels in that window. If, however, the center pixel is not an impulse, then the value of the center pixel is retained in the filtered image.

Thus, unless the pixel being considered is an impulse, the grayscale value of the pixel in the filtered image is the same as that of the input image. Thus, the Adaptive Median Filter solves the dual purpose of removing the impulse noise from the image, and reducing the distortion in the image.
Adaptive Median Filtering can handle the filtering operation of an image corrupted with an impulse noise of probability greater than 0.2. This filter also smoothens out other types of noise, thus, giving a much better output image than the standard median filter.

7.7 MOVING WINDOW ARCHITECTURE

In order to implement a moving window system in VHDL, a design was devised that took advantage of certain features of the FPGAs. FPGAs generally handle flip-flops quite easily, but the instantiation of memory on chip is more difficult. Still, compared with the other option, off-chip memory, the choice using on-chip memory was clear. It was determined that the output of the architecture should be vectors for pixels in the window, along with a data valid signal, which is used to inform an algorithm, using the window generation unit as to when the data is ready for processing. Since it was deemed necessary to achieve the maximum performance in a relatively small space, FIFO Units specific to the target FPGA were used. Importantly though, to the algorithms using the window generation architecture, the output of the window generation units is exactly the same. This useful feature allows algorithm interchangeability between the two architectures, which helped to significantly, cut down the algorithm development time. A window size was chosen because it was small enough to be easily fit onto the target FPGAs, and is considered large enough to be effective for most commonly used image sizes. With larger window sizes, more FIFOs and flip-flops must be used, which significantly increases the FPGA resources used. Figures 7.3 and 7.4 show a moving window architecture and method of reading pixel form window, used for this design for a given output pixel window.
Figure 7.3 Moving Window Architecture

Figure 7.4 Reading Pixels from Window
The rank order filter was the first algorithm to use the window_3x3 pixel window generator. Since its operation is fairly simple, it was an ideal choice. As discussed above, the rank order filter must first sort the pixel values in a window in the ascending (or rank) order. The most efficient method of accomplishing this is with a system of hardware compare/sort units, which allow for sorting a window of nine pixels into an ordered list for use in the rank order filter.

The structure was implemented as found in Figure 7.5. This system results in a sorted list after a latency of 14 clock cycles. Since the design is pipelined, after the initial latency the system produces a valid sorted list on every clock cycle. The VHDL algorithm, which implements this design, sort_3x3.vhd, is really just a series of registers and compares, as is shown in Figure 7.5. Every r(xx) box is a register and every c(xx) box is a compare unit, consisting of a simple decision. This design is accomplished quite simply in VHDL by using the following if/else statement:

```vhdl
if wx1 < wx2 then
    cx1_L <= wx1;
    cx1_H <= wx2;
else
    cx1_L <= wx2;
    cx1_H <= wx1;
end if;
```
After the sorted list is generated with the VHDL entity sort_3x3, the algorithm describing the rank order filter functionality, ro_filt_3x3.vhd, can operate on the list to produce its output. As discussed above, the rank order filter outputs a pixel value in the origin location as specified by the rank of the filter. In order to do this properly, a counter must be used to tell the output data-valid signal when to change to its ‘on’ state. Since it is desired that the output image be the same size as the input image, and the use of the
window generator effectively reduces the amount of valid output data, borders with zero value pixels must be placed around the image. In order to do this properly, counters are used to tell the algorithm when the borders start. A VHDL counter was written to count the pixel movement as the data streams into the entity. Since images are two-dimensional data, two counters were needed: one to count the rows and one to count the columns in the image. The VHDL entity that implements this functionality is called rc_counter.vhd and is found in Appendix B. Since it is a separate VHDL entity, this counter was usable in later algorithms, where this functionality was also needed. In order for the rank order filter to work properly, all three of these VHDL entities must be instantiated within the algorithm itself. This is done with standard VHDL component statements and port maps.

7.9 RESULTS AND DISCUSSION

The implementation and testing of the median filter is done considering the rectangular region. Figure 7.6 shows the results of filtering with a 3X3 median and a conditional median filter. From left to right the first row is the original Image followed by the noisy image. The second row is the output from the standard median filter, followed by the output from the Adaptive median filter. The adaptive filter works on a rectangular region $S_{xy}$. The adaptive median filter changes the size of $S_{xy}$ during the filtering operation, depending on certain criteria as listed below. The output of the filter is a single value, which replaces the current pixel value at $(x, y)$, the point on which $S_{xy}$ is centered at the time. The following notation is adapted for the simulation.

$$Z_{\text{min}} = \text{Minimum gray level value in } S_{xy}.$$  
$$Z_{\text{max}} = \text{Maximum gray level value in } S_{xy}$$  
$$Z_{\text{med}} = \text{Median of gray levels in } S_{xy}$$
\[ Z_{xy} = \text{gray level at coordinates (x, y)} \]

\[ S_{\text{max}} = \text{Maximum allowed size of } S_{xy} \]

The adaptive median filter works at two levels, denoted Level A and Level B as follows:

Level A:
\[ A1 = Z_{\text{med}} - Z_{\text{min}} \]
\[ A2 = Z_{\text{med}} - Z_{\text{max}} \]
If \( A1 > 0 \) AND \( A2 < 0 \), Go to level B 
Else increase the window size 
If window size \( \leq S_{\text{max}} \) repeat level A 
Else output \( Z_{xy} \).

Level B:
\[ B1 = Z_{xy} - Z_{\text{min}} \]
\[ B2 = Z_{xy} - Z_{\text{min}} \]
If \( B1 > 0 \) And \( B2 < 0 \) output \( Z_{xy} \)
Else output \( Z_{\text{med}} \).

The algorithm has three main purposes:
To remove ‘Salt and Pepper’ noise
To smoothen any non-impulsive noise

To reduce excessive distortions, such as too much thinning or thickening of object boundaries. The adaptive median filter is designed to remove impulsive noise from images. Therefore, our algorithm performance was first tested with basic salt and pepper noise, with a noise density of 0.25. The next test involves processing images that contain impulsive and/or non-impulsive noise. It is well known that the median filter does not provide sufficient smoothening of non-impulsive noise. Therefore, Gaussian and ‘salt and pepper’ noise were added to the image, which was then processed by the algorithm.
The work has presented a novel approach to digital image filter design based on the technique of evolvable hardware. The FPGA model for the function level evolvable hardware is analyzed and associated with the evolutionary algorithms employed. The evolution time has been greatly reduced by implementing the evolutionary algorithm in hardware. The EHW architecture evolves filters without a priori information, and out-performs a conventional filter in terms of computational effort, filtered output signal and implementation cost.

This work is the preliminary requirement to implement adaptive median filter in EHW. Before, applying EHW for this application it is important to understand the implementation difficulties in FPGA. By this work, the structure for implementing adaptive median filter in FPGA was understood and may be applied for using EHW.