2011 Sixth IEEE International Symposium on Electronic Design, Test and Application

Image Border Management for FPGA Based Filters

Donald G Bailey
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
D.G.Bailey@massey.ac.nz

Abstract— **Most of the literature on image filtering using FPGAs focuses on the normal case when the window is completely within the image. The exception - when the window is partly off the edge of the image - is rarely considered. If not managed appropriately, handling these exceptions can take up more resources than the main operation. Efficient techniques are presented that manage the image borders by reusing the logic for the standard case. A technique is described for overlapping the priming and flushing phases at the end of row and end of frame that reduce the overhead in time critical applications.**

Keywords-image processing, window filters, pipeline

I. INTRODUCTION

Real-time processing is difficult to achieve on a serial processor, particularly in an embedded environment where the clock speed must be kept low to control power consumption. For this reason, FPGAs are increasingly being used to accelerate image processing applications. FPGAs are able to exploit parallelism inherent within images by implementing significant sections of the algorithm using parallel hardware. Appropriate use of parallelism can also alleviate memory bandwidth problems associated with continual reading and writing of data to and from memory. An FPGA implementation is particularly effective for low-level image to image operations commonly associated with preprocessing. Such operations tend to be very regular (in terms of applying the same function throughout the image, and with straight-forward control logic) with relatively simple processing required for each output pixel. They are characterized, however by large volumes of data, making a conventional serial processor ineffective.

One common preprocessing operation is image filtering. A local filter produces as it output some function of the local pixel values within a window in the input image, as shown in Fig. 1. The window is scanned, with each possible window position producing a corresponding pixel value in the output.

A problem occurs with generating output pixel values on the borders of the image. In this case, the input window does not fit completely within the input image. These exceptions require additional code (in software) or logic (in hardware) to provide appropriate processing. In software, having extra code to manage these exceptions presents few problems. In hardware, however, the additional circuitry that is constructed to process the border pixels is only used occasionally, and is therefore underutilized.

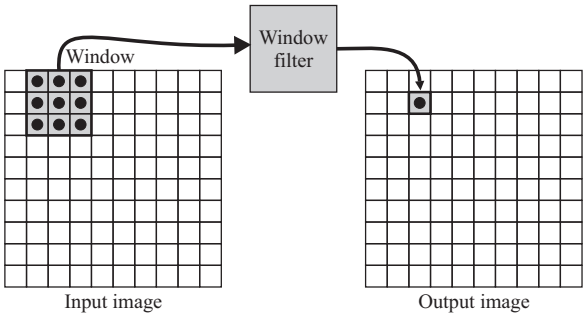

Figure 1. Operation of a window filter.

A. Outline

While there are several papers describing filter design for FPGA, very few discuss the issues associated with image borders. This paper will address this limitation systematically. In section II, the efficient implementation of filters on FPGAs will be reviewed, concentrating on the normal processing case for pipelined processing of streamed images. The different possible approaches for managing borders will be described in section III, with consequent implications for window processing. Techniques which keep the logic small, and reduce the pipeline priming and flushing latency will be presented. The FPGA resource requirements of the different schemes are compared section IV.

This paper makes two primary contributions to the field. First it systematically reviews the different border handling methods in the context of FPGA implementation. It then presents a novel border management technique that minimizes additional hardware costs and avoids the additional delays that are typically introduced by priming the processing pipeline at the end of each row and frame.

II. FPGA FILTER IMPLEMENTATION

A naïve implementation of a filter would read each pixel value from memory for each window position. Consider a simple 3×3 filter. This would require nine memory accesses to the frame buffer to obtain the input data for each output pixel calculated. Also, each pixel would be read nine times, because it appears as an input pixel for nine different window positions. (This appears to be the approach taken in [1].)

However, if the window is scanned through the image in a raster scan, there is significant overlap between window

978-0-7695-4306-2/11 $26.00 © 2011 IEEE
DOI 10.1109/DELTA.2011.34

positions from one output to the next. Pixels adjacent horizontally can be buffered and delayed using registers, reducing to three the number of pixels that must be read for each position of the 3×3 window. Since we also know that a pixel will be used on three successive rows, the pixel values may be cached within on-chip memory. For a 3×3 window it is necessary to buffer two complete rows, as shown in Fig. 2. The row buffer is basically a shift register, but is usually implemented as a FIFO or circular memory buffer.

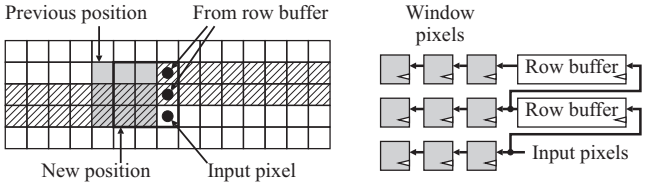

Figure 2. Row buffering.

Therefore, rather than scan the image through the window, the window is effectively fixed, with the data streamed through. Since only one pixel is loaded into the FPGA for each output pixel, each window position only requires one clock cycle. This allows data to be processed directly as it is streamed from a camera, or to a display. If necessary, the filter processing (and indeed any other image processing operations) may be pipelined to enable a throughput of one pixel per clock cycle.

If the memory or input bandwidth allow multiple pixels to be loaded in a clock cycle, then it is possible to process multiple output rows simultaneously [2]. This exploits the fact that windows in adjacent rows have many pixels in common. Depending on the particular filter function, this may also allow some of the filter logic to be shared between windows [3]. However, such enhancements will not be considered further here. They are a relatively simple extension to the techniques described later.

When processing streamed data, it is necessary to load several input rows before a pixel can be output. For a $W \times W$ window, it is necessary to have W-1 complete rows loaded plus an additional W pixels before the first pixel can be output. This is the time required to prime the processing pipeline. The windowing latency, the time from when a pixel is input to when all of the pixels are available within a window to produce the corresponding output is half of this.

III. Image Border Issues

The introduction in the previous section only considered the normal case of processing, when the window is completely within the input image. However, for pixels near the border of the image, the window extends past the edge of the input image.

A. Border handling methods

There are several different approaches to manage the case where the window extends past the edge of the image. These are reviewed here and illustrated in Fig. 3.

1) Smaller output image

One approach is to produce output pixels only where the window fits completely within the input image. As a result, the output image is smaller than the input. With a small window size, this is not usually a problem in many applications. If the smaller output is not desirable, the uncalculated pixels may be given a value by setting the value to the nearest calculated value, or other extrapolation methods. However, better results are generally obtained by extending the input image rather than the output.

2) Modify filter function

The filter function could be modified to work with the available data. For example, when smoothing noise using an averaging filter, the weights for pixels within the input image could be renormalized so that they sum to 1. For some filters, however, this is not an option because there may not be an appropriate modified function with the truncated window. Where it is used, the logic requirements of an FPGA can grow rapidly to manage the different cases.

The other alternatives are to extend the input image by inventing pixel values for the parts of the window which is outside the input.

3) Constant extension

The simplest extension of an image is with a constant value. This is often black or white, but can be any predefined value depending on the expected input and the filter function. The limitation of this is that for many images there will be a discontinuity with the contents of the image, resulting in filtering artifacts. This approach has been used in an FPGA implementation to extend an image with zeroes [1].

4) Stream processing flow

From an FPGA implementation perspective, the simplest and lowest cost approach is to completely ignore the problem of image boundaries. Processing the streamed input will assume that the first pixel of the next line immediately follows the last pixel of the current line. Similarly, the first pixel of the next frame follows the last pixel of the current frame. Since opposite edges of the image are not likely to be related, the filtered values near the borders are unlikely to be sensible. The advantage of this approach is that it requires no additional logic, and there is no additional delay to reprime the filter at the start of each row and frame or flush the filter data at the end of each row and frame.

5) Periodic extension (wrapping)

A more correct way of wrapping the image is to assume that the image is periodic both horizontally and vertically. This is the approach taken when performing an FFT for example [4]. The limitations are the same as those described earlier. A further limitation from an FPGA implementation perspective is that it would be necessary to have the last rows of the image available to be able to produce an output for the first rows. This makes a stream processing implementation impractical.

6) Duplicate borders (nearest neighbour extrapolation)

Many of the problems associated with a constant extension may be alleviated by extending the image by duplicating the outermost available pixel. This is equivalent to nearest neighbour extrapolation – selecting the nearest available pixel

value for those outside the image. This approach has been used for implementing rank filters on an FPGA [5].

7) Mirror with duplication

Rather than simply duplicate the border pixels, the pixel values outside the image can be created by mirroring those within the image. If the mirror is placed just outside the image, then the border pixels are duplicated. This approach has been used with adaptive histogram equalization [6].

8) Mirror without duplication

An alternative position of the mirror is in the middle of the border pixels. As a result, the pixel values immediately outside the input image come from 1 pixel in, and so on. The pixel values are mirrored, but without duplication of the pixels on the border of the image. This approach has been used for linear filtering on FPGAs [7].

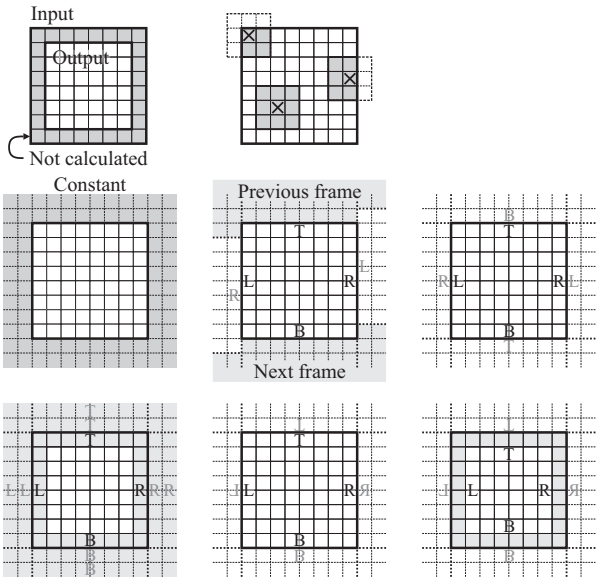

Figure 3. Comparison of border management methods. Top left: the output image is smaller; Top centre: modify filter function; Middle left: constant extension; Middle centre: default stream flow; Middle right: periodic extension; Bottom left: duplicate border pixels; Bottom centre: mirror border pixels (with duplication); Bottom right mirror border without duplication.

In this paper we will assume that a full sized output image is required. Therefore the first scheme (shrinking the output image) will not be considered. Changing the filter function has limited practicality in an FPGA implementation, and the periodic extension does not fit well with stream based processing. The edge artifacts introduced by the default stream flow are undesirable, and need to be improved on.

This leaves the remaining methods: constant extension, border duplication, and the two forms of mirroring to be considered further. It is desirable, if possible, to reproduce the performance of the default stream processing while minimizing the additional logic required. Without loss of generality, it will be assumed that the window size, W, is odd. This makes the window balanced about the output pixel position. The techniques described can be extended to even window sizes.

B. Managing borders

To avoid excessive logic, it is important that the filter function remain unmodified. This requires that the borders be managed by the block that forms the window, rather than within the filter function itself. In this way, the filter function just processes the data given to it, and makes the border management independent of the filter.

1) Direct window input

With stream processing, the obvious approach is to augment the stream with the additional data required. For example, with border duplication, the first and last rows would be loaded (W+1)/2 times, as would the first and last pixel on each row. For the mirroring methods, it would be more complex as the required rows would need to be loaded in the correct order.

This simple approach requires an additional W-1 clock cycles between rows and W-1 rows between frames to flush the old data out of the window registers and replace it with new pixel values.

There are two problems with this approach. First, if the image is being streamed from memory, the address generation logic becomes quite complex, particularly for the mirroring methods. Second, if the image is being streamed from a camera, the pixels are not used in the order provided, there are additional delays associated with priming the filter.

2) Cached priming

If the pixel values are cached as they come into the FPGA, reloading the pixel values is unnecessary. This allows a simple linear addressing, or direct processing of the streamed data as it comes directly from the camera.

First, consider the processing at the end of a row. After the last pixel in the row has been loaded, it is necessary to process and load another (W-1)/2 pixels to complete the output row (this is independent of where the extra pixels come from; whether constant, duplication or mirroring). During this time, the input stream must be stalled.

Then, to load the beginning of the next row, the filter processing must stall for (W-1)/2 pixels while the old contents of the window registers are flushed and sufficient pixels are loaded to initialize the window for the first window position. Specific details for this are shown in Fig. 4 for a 5 pixel wide window. The numbers within the multiplexers represent the input pixel number on the row.

For the constant extension, after the last pixel (N and N+1) the constant is shifted into the window registers. When beginning the next row, the first two registers already have a constant loaded, so simply shifting in the first 3 pixel values is sufficient to prepare the window for the next row.

With edge duplication, the last pixel loaded is simply fed back and shifted in again to complete the row. When the first pixel of the next row is loaded (pixel 0), it is loaded into the first 3 window registers. Then as the next pixels are shifted in, the window is prepared for the next row.

The mirroring schemes are only a little more complex. At the end of the row, the appropriate pixels from the window

registers are fed back to give the required duplication. Then as the new row is started, then in addition to clocking in the pixels at the start of the window register chain, the pixels are also clocked in to other registers as required by the mirroring.

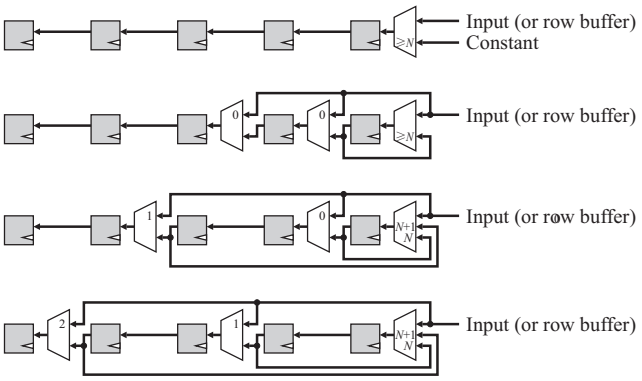

Figure 4. End of row processing for the different border management schemes (W=5). From the top: constant extension; border duplication; mirroring with duplication; and mirroring without duplication.

The arrangement for buffering rows takes the same form as that for the window registers. The only difference is that each register in Fig. 4 is replaced by a row buffer. The example shown in Fig. 5 is for mirroring with duplication. The numbers within the multiplexers represent the row number that is being loaded. After the last row of a frame is loaded, the mirrored rows are obtained from row buffers, rather than reloaded into the FPGA. At the start of the new frame, the first two rows are provided as input to multiple rows of window registers (and associated row buffers) as required by the mirroring scheme. This technique can easily be adapted to the other border management schemes.

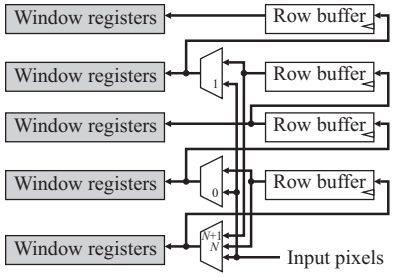

Figure 5. Row buffer arrangement for mirroring with duplication.

The effect of caching is to reduce the delay at the end of the row from W-1 clock cycles to (W-1)/2. In a similar way, the delay between frames can be reduced to (W-1)/2 row times.

3) Overlapped priming and flushing

Since there is the same number of output pixels as input pixels, it should be possible to perform the caching in such a way that requires no additional delays between rows or between frames.

Again, consider the row processing first. There are two potential sources for the (W-1)/2 clock cycle delay. The first is the time required to flush the data out of the window registers after the last pixel arrives, and the second is the time required to prime the window registers with the new data. The flushing step cannot be avoided, because of the latency associated with the filter. It is necessary to continue clocking until the last output pixel for a row is produced.

The priming step, however is unproductive. During window priming, no output pixels are being produced. However, priming cannot be avoided either, because it is necessary to load window registers before the output can be calculated.

The key to eliminating the delay is to perform the priming and flushing in parallel. Obviously the same registers cannot be used, because they still have valid data while producing the last pixels in a row. Therefore, an additional (W-1)/2 temporary registers are required to hold the data for the next row while the current row is being flushed. Then the new pixel values are loaded from the temporary registers into the window registers in parallel. This is shown for the four border management schemes in Fig. 6. There the unshaded registers are the temporary registers. The numbers within the multiplexers represent the column numbers being loaded. Here, column N is the column after the end of the row during flushing (column 0 is being loaded into the temporary registers) and N+1 is the next column (column 1). On column 2, the window register data for the previous row is replace with the data for the new row.

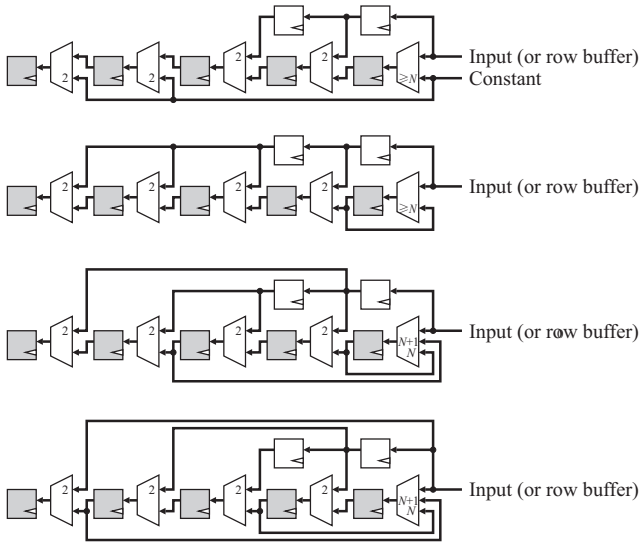

Figure 6. End of row processing for no delay for different border management schemes (W=5). From the top: constant extension; border duplication; mirroring with duplication; and mirroring without duplication.

In addition to the temporary registers, each window register requires a multiplexer on its input to enable a parallel load of the preloaded data at the start of each row.

As in the previous section, the same technique can be used for row processing by replacing the registers with row buffers. Unfortunately, the temporary registers also need to be replaced with row buffers, requiring an additional (W-1)/2 buffers. Since these use on-chip memory, this is a valuable resource.

In the previous section, the same row of data was held in multiple row buffers during flushing at the end of the frame

during priming at the start of the next frame. This duplication is unnecessary.

Instead, if the streamed data is simply fed directly into the chain of row buffers there is no duplication, and all of the data required for each row of window positions is available. This therefore requires a multiplexer on the input of each window register chain to route the output from the appropriate row buffer into each set of window registers during the transition between frames. The arrangements for the different border management schemes are shown in Fig. 7. The window register chain is the corresponding circuit from Fig. 6.

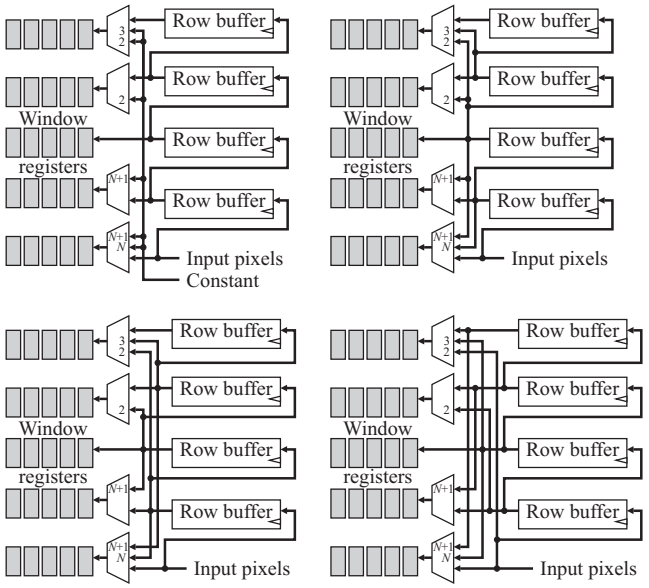

Figure 7. Row buffer arrangements for zero delay. Top left: constant extension; top right: border extension; Bottom left: mirroring with duplication; Bottom right: mirroring without duplication.

The numbers within the multiplexers correspond to the row number being loaded. Row *N* is the row after the end of the image (row 0 of the next frame), and row *N*+1 corresponds to the input of row 1. These correspond with routing the window extension during the flushing stage. Those labeled 2 and 3 are selecting the image extension when processing the next frame.

The multiplexers are therefore the same, regardless of the border management scheme used. The only difference between the schemes is the source of data.

IV. Resource Requirements

To compare the different border management schemes, a 5×5 window filter was implemented using Handel-C on a Virtex 5 FPGA. The Handel-C code was compiled to EDIF, with all optimizations turned on. The EDIF net-list was mapped to the FPGA using ISE 10.1, optimizing for area.

The filter function is not important for this comparison (it was identical for the different methods compared), and was simply the average of the pixel values within the window. Four different border processing methods were implemented for comparison:

- *Stream*: used the stream processing flow. This did no border management, and is the simplest. It is the baseline configuration.
- *Memory*: used memory based priming. Multiple accesses were made to memory to read the required additional border pixels.
- *Cached*: used the cached priming method described earlier, where each pixel is only read once and cached.
- *Overlapped*: used the overlapped priming and flushing method described in the previous section.

For each of the last three methods, four border extension schemes were implemented for comparison:

- *Constant*: window pixels outside the image were set to 127.
- *Duplicate*: the nearest edge pixels within the window were duplicated.
- *Mirror-D*: the mirroring with duplication scheme was used.
- *Mirror-N*: the mirroring without duplication scheme was used.

These 13 combinations are compared in Table I. For each configuration, the number of slice registers and number of slice LUTs reported by ISE are recorded. Also recorded are the number of clock cycles to process a 640×480 image to show the overheads of priming and flushing for the difference methods.

TABLE I. Comparison of the different border management schemes. The two numbers provided for each scheme are number of slice registers and number of slice LUTs.

	Constant	***Duplicate***	***Mirror-D***	***Mirror-N***	**Clock cycles**
Stream	316 / 287				294720
Memory	322 / 309	322 / 308	335 / 325	325 / 320	311696
Cached	338 / 302	339 / 370	339 / 380	339 / 381	309444
Overlapped	449 / 440	396 / 478	404 / 517	402 / 514	294720

The stream processing flow is the simplest and smallest, because it makes no effort to do anything about the border pixels. The image simply flows from one edge of the image to the other, and no additional processing or time is required. However border pixels of the output image are likely to be meaningless.

In this, the baseline configuration, 200 flip-flops (5×5×8 bits) are used to form the window. An additional 70 flip-flops are used within the filter code to pipeline the computation of the filter output. The remainder of the flip-flops are used for maintaining row and column addresses and other control logic. A significant fraction of the slice LUTs will be used to calculate the average pixel value of the 25 pixels within the window, with the remainder used to implement row and column counters and other control logic.

The memory reading method, while correctly managing the border pixels takes the longest because the border pixels have

to be read multiple times. Written carefully, the additional logic resources are modest. This method is only applicable when reading from a frame buffer. If directly processing streamed data (either from a camera or the output of another image processing operation) this method is inappropriate.

Caching the border pixels overcomes this problem, and also reduces the time overhead. The additional resources are primarily for the multiplexers required to direct the cached data to the appropriate window register. There is also a small amount of control logic required to detect the border pixels depending on the border extension scheme.

The overlapped processing method eliminates the time overhead, at the expense of additional resources. 80 additional flip flops (five rows by two columns by eight bits) are required to hold the window data during the overlap. This accounts for most of the additional registers required. Overlapped processing also requires more multiplexers, hence the increased logic requirement.

In terms of the different edge extensions, there is little difference between the two mirroring schemes, although both are slightly more complex than simple duplication. Simply extending with a constant is the lowest cost extension scheme, although this result is not unexpected.

V. Summary and Conclusions

This paper has reviewed the many different methods of managing image borders filtering image. The different schemes are discussed in the context of FPGA implementation, with a particular focus on pipelined processing of streamed image data. Several of the methods are impractical in this context, either because the output image is smaller than the input, or the resource requirements are excessive. This left five border management schemes: doing nothing, constant extension, border duplication, mirroring with and without duplication. The first of these is obviously the most efficient, but will give meaningless results on the borders of most images.

It was shown that directly streaming in the data as it was needed introduced a delay of W-1 clock cycles between rows and W-1 rows between frames. Memory addressing is also more awkward, particularly when mirroring is used.

This issue was addressed by caching the data so that it only needed to be loaded once. When it was needed subsequently, the appropriate data was recirculated from the cache. This simplified the addressing at the cost of a small number of multiplexers and reduced the row and frame delays to (W-1)/2 pixels and rows respectively.

Finally, it was demonstrated that the delays could be eliminated completely by overlapping the flushing and priming phases. Within the window registers, data for the next row is loaded into temporary registers while the end of the row is flushed. Then the preloaded data is loaded into the window registers in parallel, switching immediately to the start of the next line. Duplication within the row buffers was eliminated by simply streaming the input data directly into the chain of row buffers. The appropriate data was then routed to the window registers according to the border management scheme.

The cost of implementing this novel border management scheme for a $W \times W$ window is $W(W\text{-}1)/2$ temporary registers, and W^2+W-1 multiplexers regardless of the border management scheme used. This is a very modest cost and results in an efficient design.

The techniques described here can just as easily be applied to accelerated systems where several pixel values are loaded in parallel at each clock cycle.

References

[1] J.M. Ramirez, E.M. Flores, J. Martinez-Carballido, R. Enriquez, V. Alarcon-Aquino, and D. Baez-Lopez, "An FPGA-based architecture for linear and morphological image filtering," in *20th International Conference on Electronics, Communications and Computer (CONIELECOMP)*, Cholula, Mexico, 2010, pp. 90-95.

[2] B.A. Draper, J.R. Beveridge, A.P.W. Bohm, C. Ross, and M. Chawathe, "Accelerated image processing on FPGAs," *IEEE Transactions on Image Processing,* vol. 12, pp. 1543-1551, 2003.

[3] L.E. Lucke and K.K. Parhi, "Parallel structures for rank order and stack filters," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-92)*, San Francisco, California, USA, 1992, pp. 645-648.

[4] R.N. Bracewell, *The Fourier transform and its applications*, 3 ed. New York: McGraw Hill, 2000.

[5] C. Choo and P. Verma, "A real-time bit-serial rank filter implementation using Xilinx FPGA," in *Real-Time Image Processing 2008*, San Jose, California, USA, 2008, pp. 68110F-1-8.

[6] C.W. Kurak, "Adaptive histogram equalization: a parallel implementation," in *Fourth Annual IEEE Symposium Computer-Based Medical Systems*, Baltimore, Maryland, USA, 1991, pp. 192-199.

[7] A. Benkrid, K. Benkrid, and D. Crookes, "A novel FIR filter architecture for efficient signal boundary handling on Xilinx VIRTEX FPGAs," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, Napa, California, USA, 2003, pp. 273-275.