Investigation of a Digital Camera Imaging Pipeline on the RaPiD Array[1]

Technical Report UW-CSE-01-06-06

Kevin B. Rennie
University of Washington
Seattle, WA. 98195
rennie@u.washington.edu

Abstract

RaPiD has been shown to provide high performance and low power for many computationally intensive applications. RaPiD's success with these applications has been in part due to their regular structure, which leads to a homogeneous datapath and highly correlated control. To demonstrate RaPiD's ability to handle a new set of computations with a less regular structure and to further develop the architecture, we explore the implementation of a digital camera image-processing pipeline on the RaPiD array.

Introduction

RaPiD is a coarse grained reconfigurable architecture targeted for computationally intensive applications. The architecture's ability to maintain flexibility while achieving high performance at a reasonable cost has made it a strong candidate for system on a chip (SOC) solutions in the embedded and mobile application areas.

The continuing execution of Moore's Law, has propelled the digital camera marketplace in two directions. The increase in the use of CMOS sensors allows manufacturers to build lower cost, lower power solution by incorporating the camera circuitry onto the same chip. The strong yields and increasing silicon densities have made CMOS based SOC solutions increasingly attractive. On the high performance end of the spectrum, cameras using charge coupled devices (CCDs) continue to produce images with higher resolutions and increased features such as digital video. The high-end cameras will continue push envelop for performance in embedded systems. The goal of the industry to produce both low power, low cost as well as high performance digital cameras presents opportunities for reconfigurable computing solutions such as RaPiD.

This investigation of a digital camera pipeline challenges the RaPiD architecture in new ways. The computations required in an image-processing pipeline have less regularity than many of the signal processing applications we have previous considered. The imaging pipeline also requires composing several, often quite different, computations in a single datapath, another area that we have not explored in depth. These characteristics create datapath and control complexities that will test the ability of RaPiD to perform in the image-processing domain.

[1] This research was supported in part by the National Science Foundation under Grant No. 9901377.

These challenges led us to develop a novel technique for obtaining high throughput and low power for two-dimensional filtering algorithms. The digital camera pipeline has also provided the opportunity for us to look at the implementation of a complex embedded system. This paper reports the results of our evaluation of digital cameras on the RaPiD array.

The paper is organized into three sections. The first section describes our method for exploiting parallelism in any two-dimensional filtering computation. This is followed by a demonstration of the technique through the implementation of a median filter. The final section builds on the previous two by exploring the structure of a RaPiD based digital camera and concludes with results from an implementation of a real pipeline.

Pipelining a Two-Dimensional Filter

Two-dimensional filters make up a large part of image processing algorithms, so it is important to develop an efficient implementation. In the context of the digital camera, this means coming up with a method that allows high performance and low power filtering.

A description of a generic 3x3 filter helps reveal the challenges for an embedded system. The following pseudo code describes a generic 3x3 filter operating on image P of size ImageWidth by ImageHeight:

```
for (j=0; j<ImageWidth; j++) {
    for (i=0; i<ImageHeight; i++) {
        P[i][j] = filter(  P[i-1][j-1],   P[i][j-1],   P[i+1][j-1],
                           P[i-1][j],     P[i][j],     P[i+1][j],
                           P[i-1][j+1],   P[i][j+1],   P[i+1][j+1]  );
    }
}
```

While this code may be suitable for a general purpose processor, it will lead to poor results in a system with limited power resources. When described in this way, the computation of each output pixel requires N^2 memory accesses where N is the height and width of the filter, 3 in this case. The high number of memory accesses places strains the power budget as well as memory bandwidth. Furthermore, the code does not expose any parallelism that could be taken advantage of in a customized pipeline available on the RaPiD array. In order to effectively map image-processing algorithms to the RaPiD array, we developed a generic method for pipelining any NxN filter.

Our method provides the datapath with access to all pixels of the NxN filtering window on each computational cycle and reduces the required number of reads to one pixel per cycle. The key observation that makes this possible is that data can be reused between filtering windows. Visualizing the computation as an NxN window sliding around the image helps to reveal the data dependencies. For example, consider the center pixel in a 3x3 filtering window shown in the left panel of Figure 1. If the window moves down one row, as shown in the center panel, this pixel will be the upper middle pixel of a new processing window. Figure 1 shows how the pixel can be a part the same column of multiple filter windows. Similarly, when the window moves to the side, the columns of the filter windows overlap creating a dependency between columns. By sliding the window in a predictable way and buffering data appropriately, it is possible to reuse the pixels from one window to the next.

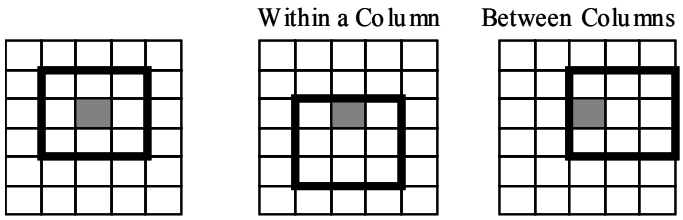

Figure 1 - Illustration of data dependency between filtering windows

Next I will develop the datapath structure for pipelining a 3x3 filter. This method can be used for any NxN filter, but a 3x3 filter will be used for purpose of illustration. In the example, I have chosen to process the image in column major order from left to right, i.e. the filtering window slides down and then across. This choice is arbitrary; the image could be processed in row major order with no significant difference. I will leave a discussion of edge effects that occur as the borders of the image and performance considerations until after the model has been fully developed.

Datapath Structure

The goal of reusing data between filter windows drives the structure of the datapath. By exploiting data dependencies between filter windows, our NxN filter implementation reads approximately one pixel per cycle and produces an output value every cycle (after some latency to fill the pipeline).

The first step is to take advantage of the data dependency within a column through the use of a column buffer. Each pixel will be a part of the same column three sequential filtering windows. First it will be in the bottom row of a filter window, then the middle, and finally the top row of the window. The structure shown in Figure 2 can be used to buffer one column of data in the 3x3 filter window.

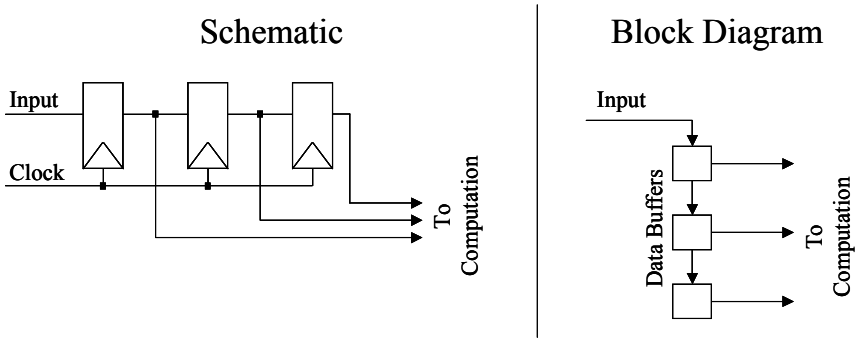

Figure 2 - Column buffer shown as a schematic and as a symbolic block diagram.

Data is serially loaded into single cycle delay buffers, or registers, and read out in parallel. Each cycle, data will advance through the pipeline as the computational window slides down the image. Three cycles after initialization, the data buffers will contain pixel values for one column in the top most filter window. On each cycle after the pipeline has been filled, the data buffers will hold the pixel values for sequential processing windows. Figure 3 shows how three column buffers can be used together to reuse data within the three columns of the 3x3 filter. The pixels in the filtering window are labeled by an abbreviation of their directional relationship to the center pixel *c*. For example, pixel *nw* is northwest of the center pixel. The required memory bandwidth for an NxN filter can be reduced from N^2 to N using N column buffers.

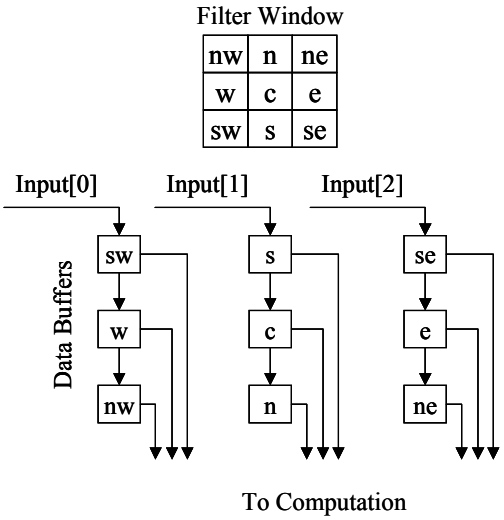

Figure 3 - Implementation of a 3x3 filter using column buffers to reduce memory reads to 3 pixels per cycle.

To further reduce memory accesses, data can also be reused between columns. As the filtering window slides down the image, it will eventually reach the bottom of the image. When this happens, it moves back to the top, over a column, and then goes back down the image. The third panel of Figure 1 illustrates the resulting data dependency between columns. In order to reuse data between columns, a memory element (RAM) that behaves as shift register is added between the column buffers. When a shift register of the correct size is used, it will store a pixel until it is ready to be used in the next column of the filtering window. Turning the column buffer sideways (to show data flowing from left to right in a conventional manner), and adding a shift register between column buffers gives the structure shown in Figure 4. When this structure is used as the pipeline for the input data, it becomes easy to expose the parallelism in the computation.

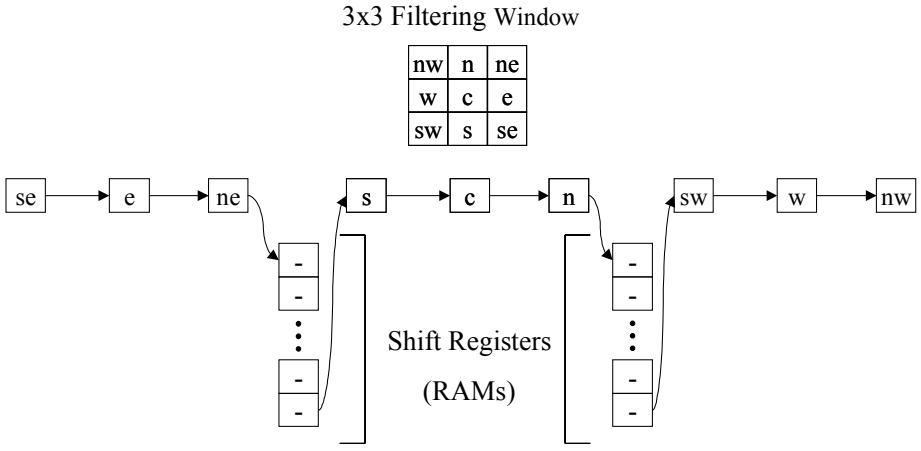

Figure 4 - Input pipeline structure for a 3x3 filter.

Each cycle a new data item will enter the pipeline from the left, and pixels will advance one position in the pipeline in the direction indicated by the arrows. On each cycle, the nine labeled registers will hold the values of sequential filter windows.

This result has two desirable effects. First, the availability of all pixels of a filter window on successive cycles provides a basis for a high throughput implementation. Second, the pixels of the filter window will always be in the same registers i.e. the southeast pixel of a filter window will always be in the first register in the pipeline. This allows the computation to be decoupled from the complexity of accessing the data. In

a sense, the filter computation can treat the registers as direct input and not have to worry about how the data got there. Thus, this pipeline structure provides an efficient method for exploiting parallelism in any NxN filter.

To help represent designs using this technique I use a simplified representation of the pipeline shown in Figure 5. This format combines the registers of the column buffer in a cluster and leaves out the shift registers between column buffers.

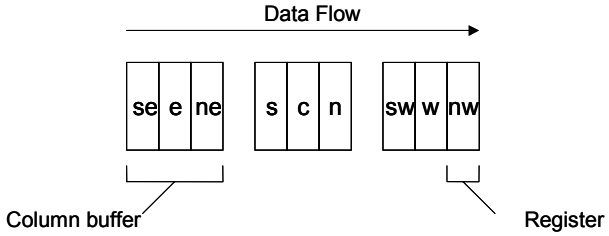

Figure 5 - Simplified block diagram of the input pipeline for a 3x3 filter.

Practical Considerations

While the structure in Figure 4 promises to reduce the number of times each pixel is read to one, this may not be possible or desirable. In order to process the whole image in one swath, the length of the shift registers used to buffer data between columns must be approximately equal to the height of the image. As a result, the size of the RAM scales with the height of the image and can require a RAM with thousands of entries for a high resolution images. The larger the memory, the more area it takes and the slower the access time. Furthermore, the RaPiD datapath does not support large memories within the datapath. Consequently, we fix the size of the shift register and introduce the notion of processing strips.

When the image is divided into horizontal strips, the size of the shift register is no longer tied to the height of the image. Instead, the height of the processing strip determines the length of the shift register. In order to process a strip, the datapath must be able to buffer the full height of the strip. This means that the maximum height of the strip is the sum of N, the filter size, and the height of the processing strip. For example, in the RaPiD benchmark architecture the datapath RAMs have 64 entries, so we process the image in horizontal strips of heights up to 64+N. The structure of the input pipeline remains the same, only the order in which the pixels are read changes.

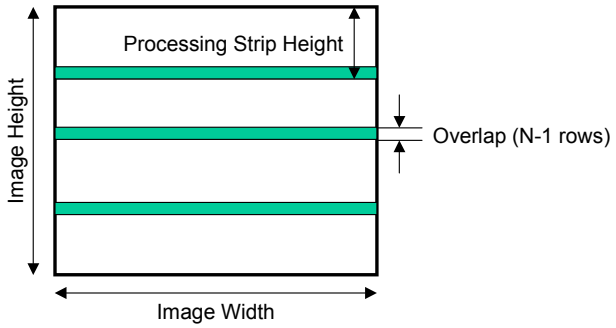

Figure 6 - An image divided into horizontal processing strips. (Scale not representative).

In the same way that the image is broken up into horizontal processing strips, the horizontal processing strips could in turn be divided into vertical strips. This allows the image to be processed in smaller pieces. This may be desirable if the memory supplying the pixel values to the datapath cannot hold the entire image. The benefits of this approach will be evaluated in a later section discussing memory models for a RaPiD based digital camera pipeline.

Edge Effects

While processing the image in strips adds benefits, it creates additional edge effects. Edge effects occur at the borders of the image where the filtering window is incomplete. For example, the filter cannot be applied to the top left pixel in the image because it has no neighbors above or to the left. In a 3x3 filter, results cannot be computed for the top and bottom rows as well as the left and right most columns of the image. These edge effects are inherent in the image and also exist in every processing strip. A filtering algorithm may specify special cases for the border pixels, but our model must handle the edge effects in the processing strips. In order to produce a valid result at each possible location, N-1 rows of the horizontal processing strips must overlap as shown in Figure 6. Similarly, N-1 columns of the vertical processing strips must overlap.

Performance

The performance of the model will be analyzed by comparing it to an optimal solution. I define an optimal solution to be one that processes the image in the same number of cycles as there are pixels in the image. In essence, the optimal solution takes one cycle per output pixel. Our model increases the total number of cycles by using processing strips and by introducing latency to fill the pipeline. Generally, the overhead of processing strips dominates the effect on performance. A detailed derivation of the performance can be found in Appendix A. Summaries of the results are presented in Table 1 and Table 2.

The most encouraging result is that the overhead of processing strips is relatively low, even with a modestly sized datapath memory. For a 5x5 filter, and the default datapath memory size of 64, the overhead is only 7% for horizontal processing strips and 14% for both horizontal and vertical processing strips. For comparison, consider the naïve implementation suggested the by original code sample at the introduction to this section. Using the same bandwidth to main memory of one read per cycle, this solution requires 25 cycles per output pixel, an order of magnitude worse than the optimal solution.

Table 1 – Ratio of total processing cycles when using horizontal strips versus an optimal solution for various filter sizes and strip heights.

		Filter Size (N)		
		3	**5**	**9**
Strip Height	**32**	1.067	1.143	1.333
	64	1.032	1.067	1.143
	256	1.008	1.016	1.032

Table 2 – Ratio of total processing cycles when using horizontal and vertical strips versus an optimal solution for various filter sizes and strip widths and a fixed strip height of 64.

		Filter Size (N)		
		3	**5**	**9**
Strip Width	**48**	1.077	1.164	1.371
	64	1.065	1.138	1.306
	128	1.048	1.101	1.219

By buffering data locally in registers and small, distributed memories, the described method achieves near optimal throughput while reading one pixel per cycle. The method can be applied to a custom datapath in an ASIC or an FPGA and is particularly well suited for the implementation of the digital camera imaging pipeline on the RaPiD array. The next section will demonstrate this with a median filter.

Median Filter

A median filter has a smoothing effect, which can be used in a digital camera pipeline to correct sensor errors. A median filter compares a pixel with its nearest neighbors to gauge whether the pixel value is reasonable or not. If the value of the comparison pixel exceeds the maximum of its neighbors by a given amount, it may be considered a defect. A similar comparison can be made to the minimum neighboring value. Depending on the type of filter the pixel found to be defective might be replaced by the minimum, maximum, or median of the surrounding pixels. For the purpose of demonstration, I have chosen to implement the most computationally challenging filter, which replaces defective pixels with the median of its neighbors. I will describe the implementation and results of using the median filter as the first stage in a digital camera image-processing pipeline.

Implementation

The implementation of the median filter on RaPiD can be described in three parts. The first step exposes the parallelism in the computation through the application of the input pipeline model developed in the previous section. The next step combines the input pipeline with the calculation of the median, maximum and minimum values. The final portion of the implementation explores opportunities and challenges for resource sharing between the red/blue and green computations.

The Bayer pattern, shown in Figure 7, dictates the shape of the median filter window. Cameras use this pattern because the sensors only detect one of the three primary colors of light. There are twice as many green sensors as red or blue because the human eye is most sensitive to green light. Using more green pixels increases the accuracy of the green data and provides the appearance of a better quality image.

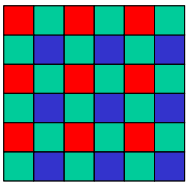

Figure 7 - Bayer matrix pattern

The median filter compares a pixel with its eight nearest neighbors of the same color to determine if it should be replaced. In the green computation, the neighboring eight pixels form a diamond around the center pixel. For the red and blue cases, the nearest surrounding pixels form a square around the center pixel. Since the filter window for red and blue pixels is identical, they can be treated as one color for the purpose of describing the algorithm. Figure 8 depicts the filter windows separately and shows how the two computations can be combined in single 5x5 window. Even though this may appear to be two separate filters, both filters can be implemented using the same input pipeline.

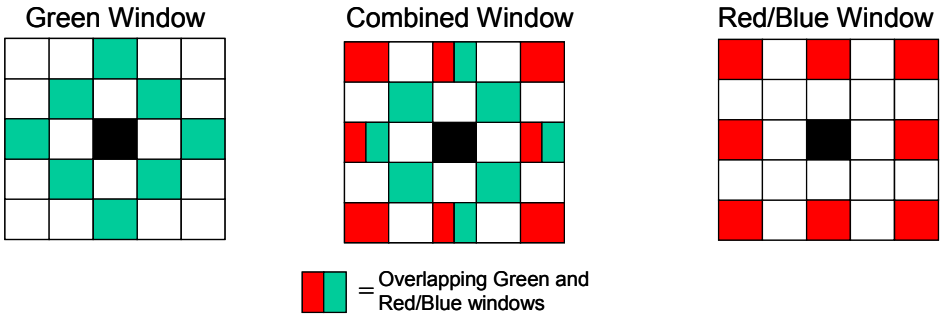

Figure 8 - 5x5 filter window for the median filter.

The method for pipelining a 2-D filter developed in the previous section can be applied to the combined filter window to produce a datapath for pipelining the computation. The block diagram in Figure 9 represents the pipeline using the format introduced in Figure 5. The transformation of the filter window in to a pipeline can be visualized by taking each column, turning it sideways, and placing them together in a row. The color coding in Figure 9 makes it clear which values will be used in the median filter computation based on the color of the center pixel. Green colored registers will be used when the center pixel is green and the red registers will be used for either the red or blue computation. Because of the Bayer pattern, the computation will switch between colors on every cycle. This pipeline sets the stage for a high throughput implementation.

Figure 9 - Symbolic representation of the input pipeline for the median filter.

The next step in the implementation combines the filter with the input pipeline. In the case of the median filter, the computation of the maximum, minimum, and median values essentially requires sorting a list of 8 pixels. Since the list has an even number of pixels, the median will be an average of the middle two values of the sorted list. A pipelined version of insertion sort provides a good match for both the application and the RaPiD architecture. Figure 10 shows the combination of an insertion sort pipeline with the input pipeline for each color. To more clearly illustrate the implementation of the datapath, the green pipeline is shown separately from the red/blue pipeline.

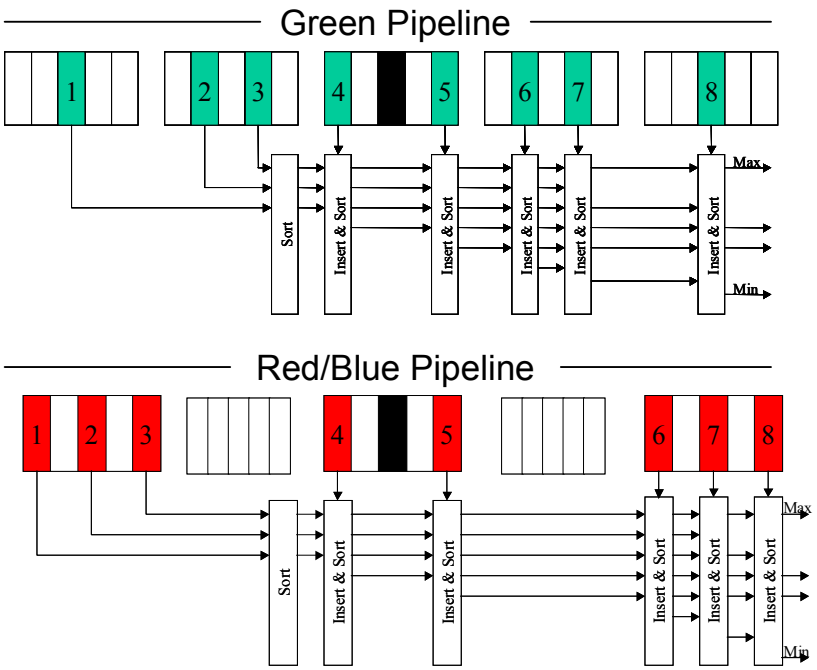

Figure 10 - Implementation of maximum, minimum and median calculation in the median filter for both pipelines.

The final step in the implementation of the median filter is sharing resources between the two pipelines shown in Figure 10. As suggested in Figure 8 and Figure 9 the input pipeline can be completely shared. Even though the filter windows require values from different registers, the column buffers and shift registers required to pipeline the input data are used in the same way by both filters. The greater challenge lies in sharing the insertion sort hardware. With sufficient communication, all functional units could be shared between both pipelines. Increased communication requires additional data buses, or pipes in RaPiD, and increases the area and complexity cost. Consequently, it is important to balance the desire to multiplex hardware resources with the increased cost of communication. In order to minimize the number of pipes, only functional units in the same stage of the RaPiD datapath were shared between computations. The horizontal position of resources in Figure 10 corresponds to the location in the datapath, i.e. resources that fall on the same vertical line occur in the same stage. This feature of the diagram quickly points out the opportunities to share functional units between the first three parts of the sorting pipeline and the stage that handles the 8th green pixel and the 7th red/blue pixel. Overall, the majority of the insertion sort pipeline can be shared between the green and red/blue filters.

Further reductions in resource usage are possible by optimizing the insertion sort computation. The median filter requires only the first, last and middle two values from a sorted list. As soon as the pipeline can detect that a value will not be one of the four desired values, it can be dropped from the pipeline. For example, in sorted a list of size 7, the 2nd largest value will not be of interest for the final sorted list. Only one value remains to be added to the list, so the 2nd largest pixel in a list of size 7 can be only the 2nd or the 3rd largest in a list of size 8. This optimization to insertion sort reduces both the number of functional units and the communication used between stages. The decrease in communication is reflected in Figure 10 by the decreasing number of buses after stage 7.

Results

Table 3 – Median filter resource requirements

Item	Quantity
ALU	40
Multiplier	0
Shifter	1
RAM	4
Control Bits	11
Max Pipe Usage	8
Memory I/O (P/cycle)	2

The resource requirements for the median filter shown in Table 3 reflect tradeoffs made to reduce communication resources. Complete sharing of functional units uses 10 pipes and 30 ALUs. With no sharing, 60 ALUs and 8 pipes are needed. The implementation described above manages to maintain the minimum of 8 pipes while reducing the required ALUs to 40. Depending on the final application it may be desirable to trade 2 pipes for 10 ALUs.

Two unexpected results from the implementation of the median filter are the small number of control bits and low memory bandwidth. At first glance, the median filter does not seem like a good candidate for RaPiD because of the apparent high memory bandwidth and complexity of multiplexing two sorting calculations. The method developed for pipelining a 2-D filter made this a successful application. It turns the irregular memory access pattern of a naïve solution into a regular column major pattern, which contributed to using only 11 of the available 31 control bits. More complex applications require a larger number of control bits. For comparison, matrix multiplication uses 9 control bits. The use of the input pipeline model also reduced the required memory bandwidth to two pixels per cycle, one read and one write.

The throughput of the median filter is nearly the same as the throughput of the generic 5x5 filter. In the final implementation, the datapath will be pipelined and retimed adding additional latency. The penalty of filling the pipeline will be small relative to the total number of processing cycles. The throughput using horizontal processing strips will be very near to 1.07 cycles per pixel, only 0.07 cycles per pixel above an optimal solution.

Power is another consideration for a median filter. We have chosen to implement the most computationally challenging correction, which replaces defective pixels with the median of its neighbors. The most intensive part of the computation is finding the median value, which may or may not be used. Computing the median only when necessary could reduce power consumption. Most likely, this would require another pass through the data and hence reduce the throughput. Nevertheless, performance can be traded for power.

The implementation of the median filter demonstrates the successful mapping of a complex image-processing algorithm onto the RaPiD array. This result will be extended in the next section to a sample processing pipeline for a digital camera.

A RaPiD Based Digital Camera Pipeline

This section describes a SOC solution based on RaPiD for a digital camera. The initial investigation involved the implementation and analysis of a digital camera pipeline from STMicroelectronics. In order to protect their intellectual property, I will propose a system solution using a generic imaging pipeline but present the results from our implementation of their pipeline. In proposing a solution for a generic pipeline, I will qualitatively examine the architecture of a RaPiD based solution and suggest two possible memory models.

Basic Architecture

The sample pipeline in Figure 11 provides a framework for developing a digital camera implementation using RaPiD [2]. The pipeline, while generic, is representative of the steps required to take data values from the camera's sensors and turn them into a final compressed image. The interpolation stage converts a Bayer matrix, where each pixel only has one color value, into an RGB image where each pixel is composed of three values, one for each primary color of light. Following interpolation, various filters are applied to the image to improve its quality. Two stages of image improvement, correction and enhancement, are shown in the same datapath configuration to represent that several parts of an algorithm may be combined into the same configuration. Lastly, the image is compressed using standard techniques such as JPEG.

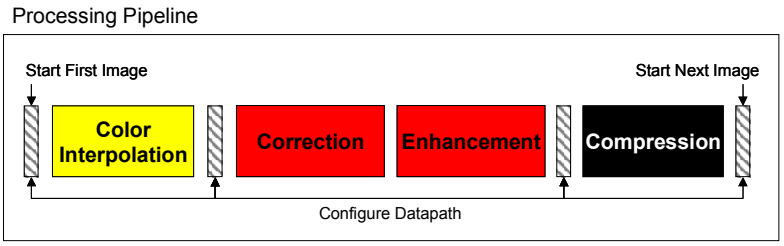

Figure 11 - Sample image processing pipeline for a Digital Camera.

The custom reconfigurable datapath provided by the RaPiD architecture performs the computationally intensive portion of the image processing represented by the sample pipeline. The datapath could be constructed to process the entire image in one configuration, but the reconfigurability of RaPiD can be leveraged to partition the algorithm into stages and reduce the size of the datapath. Figure 11 shows a possible partitioning of a generic pipeline. When the image is captured, the RaPiD datapath will be configured for color interpolation. The image will be processed in whole or in part, and the results written back to memory. Then the datapath will be reconfigured for the next stage of the computation.

Using RaPiD to build an image-processing pipeline in this way provides several design wins. First, the algorithms can be implemented in a manner that exposes fine-grained parallelism in the computations. In this way, a custom datapath will experience improved performance and reduced power consumption as compared with general purpose processor solution. This feature was demonstrated with the median filter in which a throughput of near one pixel per cycle was obtained using only 2 pixels per cycle of memory bandwidth. The techniques used with the median filter are useful in the generic pipeline as well, particularly in the color interpolation and image improvement stages. In color interpolation the size of the processing window can be as large as 9x9 [3], in which case a custom datapath will have a significant advantage over a general purpose processor. The processor must read 81 pixels to produce a single output, while our solution reads only one. ASICs or other reconfigurable solutions can also reap these same benefits. However, I expect the coarse grained functional units available on the RaPiD array will lead to higher performance than an FPGA for mathematical filtering operations.

The reconfigurability of the datapath also provides advantages over a camera built on ASICs. The ability to adapt the hardware provides a way to keep up with evolving standards and tailor the camera to a specific market without having to fabricate new hardware. These features improve the time to market and could be a critical edge over competitors. Reconfigurability also relaxes the constraint placed on ASICs that the

algorithm must be implemented in a single pipeline. Partitioning the algorithm essentially folds pipeline into a smaller piece. Depending on how well the resource requirements of the stages can be matched, there may be potential area savings for reconfiguring the same hardware. The area overhead required to provide reconfigurability will work against this benefit.

Overall, a course grained reconfigurable architecture like RaPiD offers opportunities to improve upon processor and ASIC solutions. These advantages must be balanced with the potential drawbacks such as the complexity of implementing a reconfigurable solution.

Memory Models

As a component of an embedded system, the architecture of the memory system is an important design consideration. Partitioning the imaging pipeline means that intermediate results must be stored in memory between stages of the computation. The memory accesses created by reading and storing the intermediate values places a large strain on the memory bandwidth. I will qualitatively examine the implications of a basic model that connects the datapath directly to memory as well as a second model, which uses a local memory to reduce the required main memory bandwidth.

In the most basic memory model, the datapath will interact directly with main memory. Data is streamed into the RaPiD datapath, processed, and written back to memory. Once the entire image has been processed, the datapath is reconfigured for the next computation. The data flow for this main memory model is shown in the left panel of Figure 12. In this model, the datapath will be reading and writing memory at the same time. At the input and output of the image improvement stage, each pixel will have red, green and blue values, meaning that the memory must be able to handle 6 pixel-sized transactions per cycle to maintain maximum throughput.

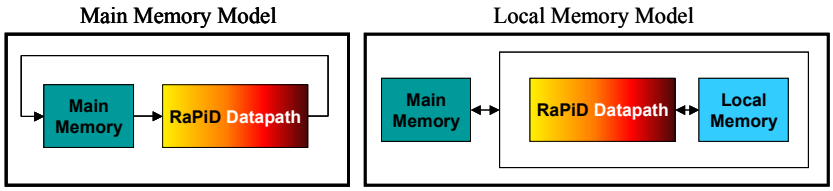

Figure 12 - Memory system models. Main memory stores the complete image in both cases.

The high peak bandwidth and redundant memory accesses of the main memory model lead to a second model which uses a small memory local to the datapath chip to buffer part of the image. Instead of performing one computation on the entire image, all computations are now performed on the portion of the image stored in the local memory before moving onto another area of the image. The flow of data for this scheme is shown in Figure 13.

The local memory model achieves a reduction in main memory bandwidth at a cost of an increased number of processing cycles. Because the main memory is only being read or written, but not both, the peak memory bandwidth is half that used in the main memory model. The average bandwidth, and in turn power, experiences greater than 50% reduction because the memory is idle during the image improvement calculation. Unfortunately, processing the image in smaller pieces requires more cycles and reduces the throughput of the pipeline. The performance penalty due to edge effect of the strips is compounded when several filters are applied to the same subset of the image. (Appendix A describes this effect in more detail). Furthermore, the datapath must be reconfigured many more times per image than in the main memory model. In order to support this model, the RaPiD datapath must provide fast context switching to minimize the reconfiguration penalty. In spite of increasing the number of processing cycles, using the local memory model moves the high bandwidth demands to a smaller on chip memory, which may have performance benefits. As an order of magnitude or two smaller than main memory, the local memory will be better able to keep up with the throughput and access time requirements of a high performance datapath.

Overall, the main and local memory models provide the opportunity to trade performance and main memory bandwidth.

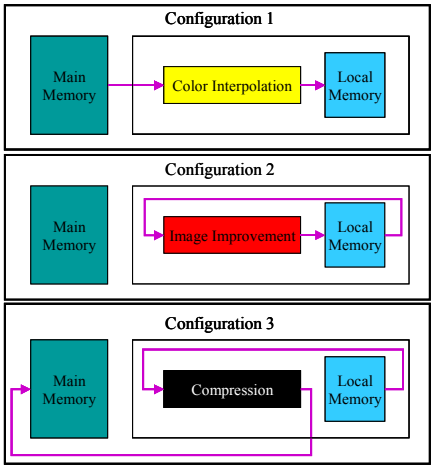

Figure 13 - Data flow for processing a subset of the image using the local memory model. These steps will be repeated several times to process the entire image.

Performance

While the specifics of the STMicroelectronics imaging pipeline cannot be disclosed, the results of the implementation quantify the performance of the proposed methodology. The STMicroelectronics pipeline was divided into three configurations, just like the sample pipeline. The results in Table 4 were derived from the results of the implementation and with the help of the equations developed in Appendix A

Table 4 - Performance Summary for the Implementation of a Digital Camera Pipeline Using 3 Memory Models.

	Main	128KB Local	16KB Local
Peak Main Memory Bandwidth (P/cycle)	6	3	3
Average Main Memory Bandwidth (P/cycle)	5.33	1.33	1.33
Cycles per pixel	3.167	3.386	3.781
Mega-pixels/second at 50 MHz	14.63	13.67	12.26

First and foremost, the results demonstrate the feasibility of RaPiD as a platform for a high performance digital camera. The results also show the effectiveness of the local memory model in reducing the main memory bandwidth at the cost of a relatively small reduction in performance. The throughput numbers do not take into account the reconfiguration time. With only three configurations per image an implementation, the main memory model will allow for longer reconfiguration without a significant impact on performance. In the local memory models, it is critical for the datapath to support fast reconfiguration. This could be supported by caching several configurations in the datapath to minimize the overhead of switching between computations.

To provide a frame of reference for the numbers, an $800 semi-pro Nikon CoolPix 990 supports resolutions up to 3.3M pixels for still images and digital video at 30 FPS for a 300K pixel image. Many additional design considerations and unknown factors prevent comparing the performance of the Nikon with the implementation of STMicroelectronics' pipeline on the RaPiD array. I provide them only to give context to the results and to reinforce the potential of RaPiD for a digital camera solution.

Conclusions

The marketplace forces in digital still cameras demand hardware solutions that can balance high performance with low power and low cost. These forces provide opportunities for coarse-grained reconfigurable architectures like RaPiD. The complexity of the image processing filters promoted the development of a new method for pipelining filters on RaPiD. This technique provided a platform for a high throughput solution that minimizes memory bandwidth and fits well within the bounds of the complexity RaPiD can handle.

The results from the implementation of STMicroelectronics' imaging pipeline demonstrate RaPiD to be a strong candidate for a high performance, low power SOC digital camera solution.

Future Work

The implementation of the digital camera pipeline raises some interesting questions for future investigation. Throughout the implementation I made an effort to reuse resources, both between configurations and within a single configuration. The local memory model depends heavily on the ability to quickly switch between configurations of the datapath. While it is certainly feasible to cache different configurations in the datapath, this is an area we have not explored in depth. The architecture must also address how to quickly switch between controlling one datapath and the next. Multiplexing the hardware within a single configuration had to be done manually. It may be possible to add support to the language to make it easier to explicitly time multiplex or an even better solution would be to have the tools automatically detect these opportunities. With the development of the emulation system, we will have the opportunity to further explore some of these topics.

References

[1] Ebeling, C.; Cronquist, D. C.; Franklin, P.; Berg, S.; "Mapping applications to the RaPiD configurable architecture", Field-Programmable Custom Computing Machines (FCCM-97), April 1997, pp. 106-15.

[2] Ping Wah Wong; Tretter, D.; Herley, C.; Moayeri, N.; Lee, R.; "Image Processing Considerations for Digital Photography", Compcon '97. Proceedings, IEEE , 1997.

[3] Curtin, D., "Understanding How Image Sensors Work", http://www.shortcourses.com/how/sensors/sensors.htm, 2001.

Appendix A - Performance Derivation for the Two-Dimensional Filtering Model

This supplement to the text derive equations for the performance of the two dimensional filtering model. The model provides a method for pipelining the computation of any NxN filter, where N is the height or width of the filter window. The derivation will analyze the implications of pipeline latency and the performance of using only horizontal processing strips as well as horizontal and vertical processing strips.

Pipeline Latency

The 2-D filtering model appears to introduce latency to the results because of the time required to fill the pipeline. Whether or not this increases the overhead above an optimal solution depends on the details of the filter and the type of processing strips used. In the general case, we can consider two types of filters, those that reduce the image size due to edge effects and those that use special cases to avoid the reduction in image size. For reducing filters, the latency does not matter i.e. the number of processing cycles is equal to the number of required reads. For non-reducing filters, the special cases for the border columns must be considered. The columns on the left edge of the image can be processed during the pipeline fill time. The columns on the right hand border of the image require additional cycles to be processed. This increased the processing burden by (N-1)/2 columns for odd size filter and N/2 columns for even sized filters. For most reasonable image sizes, these additional cycles are negligible. These claims will be quantified in the next section.

Non-Reducing Filter with Horizontal Processing Strips

Consider the image with R_i rows and C_i columns, a filter size of N, and horizontal processing strips of height R_s shown in Figure 14.

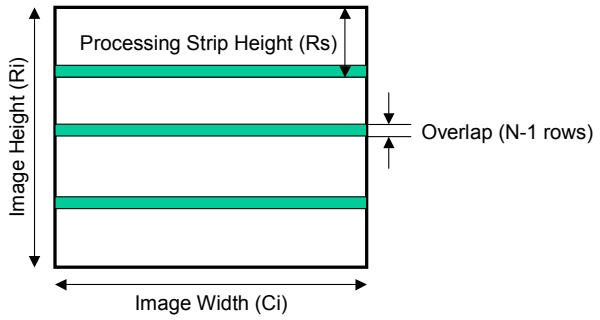

Figure 14 - Image labeled with dimensions used in performance calculations.

The performance in terms of the number of cycles per pixel can be found by first calculating the total number of cycles. This is the product of the number of strips and the number of cycles per strip which are given by:

$$Number\ of\ Strips = \frac{R_i}{(R_s - (N-1))} \qquad Cycles\ per\ Strip = R_s \times C_i + R_s \times \frac{(N - odd(N))}{2}$$

The number of horizontal strips is the total number of rows divided by the number of rows of valid results in each strip. The formula for cycles per strip is less obvious. The first term is the minimum number of cycles per strip and is equal to the number of pixels in a strip. The second term applies only to non-reducing filters and reflects the number of additional cycles for processing the border columns on the right side of the image. Depending on the filter, this may mean simply copying the columns or it may require specialized processing. Either way, (N-1)/2 or (N/2) columns of additional cycles are necessary for odd or even sized filters. The total number of cycles to process the image is the product of the number of strips and the number of cycles per strip. A performance ratio to the optimal solution can be found by dividing

the total number of cycles by total number of pixels (R_i*C_i). An optimal solution will process the image in (R_i*C_i) cycles, so the ratio will be greater than 1. These equations are shown below:

$$Total\ Cycles = \frac{R_i}{(R_s - (N-1))} \times (R_s \times C_i + R_s \times \frac{(N - odd(N))}{2})$$

$$Cycles\ per\ Pixel = \frac{R_s}{(R_s - (N-1))} \times (1 + \frac{(N - odd(N))}{2 \times C_i})$$

The formula for cycles per pixel directly gives the performance ratio. The formula is the product of the number of strips and one plus a term that captures the right side edge effects of non-reducing filters. For typical filter sizes N and image sizes C_i this term will be on the order of 1/100 or 1/1000. Therefore, edge effects only contribute around 1% to the overhead of the pipeline model.

Because the contribution of edge effects and pipeline latency can be complicated, it is worth stating how I have incorporated these factors into my calculations. By considering the performance of the image processing in terms of the number of strips and the cycles per strip, the latency of the results and the edge effects on the top, bottom, and left of the image and the strips are implicitly included in the equations. Only in non-reducing filters must additional cycles be considered. The top and bottom edge effects reduce the number of valid results per strip and are incorporated into the equation that determines the number of strips. The left side edge effects do not influence the performance because affected columns can be processed during the fill time of the pipeline.

Non-Reducing Filter with Horizontal and Vertical Processing Strips

The performance equations derived for horizontal strips can be extended to consider vertical processing strips of size C_s. The main difference is that the number of processing strips is now a function of both R_s and C_s. Furthermore, the additional cycles to account for the edge effects on the right side of the image only need to be added to the right most vertical processing strips. The resulting formulas are shown below.

$$Total\ Cycles = \frac{R_i \times C_i}{(R_s - (N-1)) \times (C_s - (N-1))} \times R_s \times C_s + \frac{R_i}{(R_s - (N-1))} \times \frac{(N - odd(N))}{2}$$

$$Cycles\ per\ Pixel = \frac{R_s}{(R_s - (N-1))} \times (\frac{C_s}{(C_s - (N-1))} + \frac{(N - odd(N))}{2 \times C_i})$$

The decrease in performance of using both processing strips is reflected in the change in the second term of the cycles per pixel formula. For horizontal processing strips, this term is 1, for both processing strips the term will be greater than one and increase as the C_s decreases. This makes intuitive sense since the overhead should increase as the size of the strip is reduced because pixels must be reread more frequently.

Complication of using Horizontal and Vertical Processing Strips

The motivation for using both vertical and horizontal processing strips comes from a scenario in which the memory connected to the datapath cannot hold the entire image. A complication arises when using this memory to store intermediate values between computations. The edge effects of filtering in each successive computation will repeatedly reduce the size of the intermediate processing area regardless of the type of filter. These effects are additive. For example, suppose the memory can hold 64x64 pixels. Applying a 5x5 filter to this part of the image will produce 60x60 valid pixels. If a 5x5 filter is applied to the intermediate result of size 60x60, the next result will be valid for only a strip of size 56x56. This additive effect increases the required overlap, and the cost of processing the image in strips using a local memory. Equations for the general case are not presented here.

Appendix B – Source Code for the Implementation of the Median Filter

Header file: med_filt.h

```
 1 ////////////////////////////////////////////////////////////////////
 2 //
 3 //             Constants and Parameters for
 4 //                  Median Filter
 5 //
 6 // Kevin Rennie
 7 // 21 March 2000
 8 //
 9 // med_filt.h
10 //
11 ////////////////////////////////////////////////////////////////////
12
13 #define STAGES 28
14
15 #include "rapidb.h"
16
17 // Ve5 Constants
18 #define FALSE 0
19 #define TRUE 1
20 #define RED 0
21 #define GREEN 1
22 #define GRN GREEN
23 #define BLUE 2
24 #define BLU BLUE
25 #define COLORS 3
26 #define VJOG FALSE
27 #define HJOG FALSE
28 #define IP_B0 10              // input U (buswdith, U = unsigned)
29 #define IP_B1 IP_B0           // output U
30 #define DEF_REPORT FALSE
31 #define DEF_SCYTHE TRUE
32 #define DEF_FILTER FALSE
33 #define DEF_ORIGIN TRUE
34 #define DEF_RANK 1
35 #define DEF_THRESH 8
36 #define DEF_MAXDEFS 12
37 #define DE_B0 IP_B1           // input U
38 #define DE_B1 10              // truncation used in scythe sorts U
39 #define DE_B2 10              // truncation used in ring_median sorts U
40 #define DE_B3 6                      // truncation used in map severity U
41 #define DE_B4 6                      // truncation used in threshold input U
42
43 // Driver Program constants
44
45 #define CONSTANTS 0
46 #ifdef BIG_TEST
47 #define IVSIZE 164            // Number of input rows
48 #define IHSIZE 104            // Number of input columns
49 #define ROWS_PER_STRIP 8      // Number of rows in each processing strip
50 #else
51 #define IHSIZE 8
52 #define IVSIZE 32
53 #define ROWS_PER_STRIP 8
54 #endif
55
56 #define LOST_ROWS 5                    //
57 #define LOST_COLS 5                    //
58 #define STRIPS (IVSIZE / ROWS_PER_STRIP)  // Number of processing strips
59
```

RaPiD-C Code: med_filt.rc

```
     1 ////////////////////////////////////////////////////////////////////
     2 //
     3 //                  Rapid Program for
     4 //                   Median Filter
     5 //
     6 // Kevin Rennie
     7 // 21 March 2000
     8 //
     9 // med_filt.rc
    10 //
    11 ////////////////////////////////////////////////////////////////////
    12
    13 #include "med_filt.h"
    14
    15 void med_filt_rapid(Word in[IVSIZE][IHSIZE], Word out[IVSIZE][IHSIZE])
    16
    17 /*
    18 void med_filt_rapid(Word in[IVSIZE][IHSIZE],
    19                     Word outA[IVSIZE][IHSIZE], Word outB[IVSIZE][IHSIZE], Word
outC[IVSIZE][IHSIZE])
    20                    */
    21 {
    22   StreamOut outStream;               // Explicitly fill output stream
    23   Pipe inData(1);                    // Delayed Input pipe
    24   Pipe inDataBroad;                  // Broadcast Input data
    25   Pipe onePipe, twoPipe, threePipe;  // Pipes for sorted list
    26   Pipe fourPipe, fivePipe;           // Pipes for sorted list
    27   Pipe pixelPipe;                    // Pipe for the center pixel
    28   Pipe maxPipe, minPipe;             // Pipe for min and max values
    29   Pipe outPipe;                             // Ouptut Pipe
    30
    31   Pipe one;
    32
    33   Bit gsiteBit;
    34   BitPipe gsite;
    35
    36   Pipe testPipeA, testPipeB, testPipeC;
    37
    38   Ram inCol1, inCol2, inCol3, inCol4;
    39
    40   // Comparison bits for insertion
    41   Bit CompOne[STAGES], CompTwo[STAGES], CompThree[STAGES];
    42   Bit CompFour[STAGES], CompFive[STAGES];
    43   Bit CompMax[STAGES], CompMin[STAGES];
    44   Bit correct[STAGES];
    45
    46   // Storage for sorting
    47   Word oneReg[STAGES], twoReg[STAGES], threeReg[STAGES];
    48   Word fourReg[STAGES], fiveReg[STAGES];
    49   Word tempReg[STAGES];
    50   Word pixelReg[STAGES];
    51   Word maxReg[STAGES], minReg[STAGES];
    52
    53   Event outReady, outWrite;
    54   Event outWrite0, outWrite1, outWrite2;
    55
    56   For c;
    57   For i, j, k;
    58   For io, jo, ko;
    59   For ioF, joF, koF;
    60   For ioM, joM, koM;
    61   For ioL, joL, koL;
    62   For isF, jsF, ksF;
    63   For isM, jsM, ksM;
    64   For isL, jsL, ksL;
    65   For is, js, ks;
    66   For q_cons, q_repeat, q_cols, q_rows;
```

```
 67   For outWait;
 68
 69   Par {
 70
 71   thread:
 72     for (q_repeat=0; q_repeat<(IHSIZE*IVSIZE); q_repeat++) {
 73       for (q_cols=0; q_cols<IHSIZE; q_cols++) {
 74        for (q_rows=0; q_rows<ROWS_PER_STRIP-5; q_rows++) {
 75          Datapath {}
 76        }
 77       }
 78     }
 79
 80   thread:
 81     for (k=0; k<=(IVSIZE-ROWS_PER_STRIP); k+=(ROWS_PER_STRIP-4)) {
 82       for (j=0; j<(IHSIZE+5); j++) {
 83        for (i=0; i<ROWS_PER_STRIP; i++) {
 84          Datapath {
 85            if ((i==2) && (j==2)) {
 86              signal(outReady);
 87            }
 88            if (s==0) {
 89              inDataBroad = in[i+k][j];
 90            }
 91            if (s==1) {                        // (25) +idip (R/B)           (RB1)
 92              // Initial site indicator
 93              //if (k.first) {
 94              if (i.first && j.first) {
 95               gsiteBit = !(VJOG^HJOG);
 96              }
 97              else if (!i.first) {
 98               gsiteBit = !gsiteBit;
 99              }
100              gsite = gsiteBit;
101              // Start delayed input pipe
102              inData = inDataBroad;
103
104              // if (!gsite)
105               onePipe = inData;
106            }
107            always if (s==2) {                 // (24)
108            }
109            always if (s==3) {                 // (23) -iinc (G & R/B)
     (G1, RB2)
110              if (gsite) {
111               onePipe = inData;
112              }
113              else {
114               twoPipe = inData;
115              }
116            }
117            always if (s==4) {                 // (22)
118            }
119            always if (s==5) {                 // (21) -idim (R/B)           (RB3)
120              if (!gsite) {
121               threePipe = inData;
122              }
123              //if ((i.first && j.first) || (inCol1.address == (ROWS_PER_STRIP-
5))) {
124              if (q_rows.first) {
125               inCol1.address = 0;
126              }
127              tempReg[s] = inData;
128              inData = inCol1;
129              inCol1 = tempReg[s];
130              inCol1.address++;
131            }
132            always if (s==6) {                 // (20)
133            }
134            always if (s==7) {                 // (19) +idipg (G)            (G2)
135              if (gsite) {
```

```
136                twoPipe = inData;
137               }
138              }
139              always if (s==8) {                // (18)
140              }
141              always if (s==9) {                // (17) -idimp (G)            (G3)
142               if (gsite) {
143                threePipe = inData;
144               }
145              }
146              always if (s==10) {                       // (16)
     (Have G3, RB3) Shared
147               // Sort one, two and three
148
149               // oneReg = max(1P, 2P); twoReg = min(1P,2P)
150               if (onePipe > twoPipe) {
151                oneReg[s] = onePipe;
152                twoReg[s] = twoPipe;
153               }
154               else {
155                oneReg[s] = twoPipe;
156                twoReg[s] = onePipe;
157               }
158
159               // oneReg = max(3P,max(1P,2P)); threeReg = min(3P, max(1P,2P));
160               if (oneReg[s] > threePipe) {    // 1R is max
161                threeReg[s] = threePipe;
162               }
163               else {
164                threeReg[s] = oneReg[s];
165                oneReg[s] = threePipe;        // 3P is max
166               }
167
168               // twoReg = max(min(1P,2P), min(3P,max(1P,2P))); threeReg =
min(min(1P,2P), min(3P,max(1P,2P)))
169               if (twoReg[s] < threeReg[s]) {  // 2R is min, switch
170                tempReg[s] = twoReg[s];
171                twoReg[s] = threeReg[s];
172                threeReg[s] = tempReg[s];
173               }
174               // 1R > 2R > 3R
175               onePipe = oneReg[s];
176               twoPipe = twoReg[s];
177               threePipe = threeReg[s];
178
179               // End of column buffer
180               //if ((i.first && j.first) || (inCol2.address == (ROWS_PER_STRIP-
5))) {
181               if (q_rows.first) {
182                inCol2.address = 0;
183               }
184               tempReg[s] = inData;
185               inData = inCol2;
186               inCol2 = tempReg[s];
187               inCol2.address++;
188              }
189              always if (s==11) {                       // (15) +ijmp (G & R/B)
     (G4, RB4) Shared
190               CompOne = inData > onePipe;
191               CompTwo = inData > twoPipe;
192               CompThree = inData > threePipe;
193
194               if (CompOne) {          // (In, One, Two, Three)
195                fourPipe = threePipe;
196                threePipe = twoPipe;
197                twoPipe = onePipe;
198                onePipe = inData;
199               }
200               else if (CompTwo) {             // (One, In, Two, Three)
201                fourPipe = threePipe;
202                threePipe = twoPipe;
```

```
203                twoPipe = inData;
204               }
205               else if (CompThree) {   // (One, Two, In, Three)
206                fourPipe = threePipe;
207                threePipe = inData;
208               }
209               else {                  // (One, Two, Three, In)
210                fourPipe = inData;
211               }
212             }
213             always if (s==12) {                     // (14)
214             }
215             always if (s==13) {                     // (13)  iptr (G & R/B)
216               pixelPipe = inData;
217             }
218             always if (s==14) {                     // (12)
219             }
220             always if (s==15) {                     // (11) -ijmp (G & R/B)
    (G5, RB5) Shared
221               CompOne = inData > onePipe;
222               CompTwo = inData > twoPipe;
223               CompThree = inData > threePipe;
224               CompFour = inData > fourPipe;
225
226               if (CompOne) {          // (In, One, Two, Three, Four)
227                fivePipe = fourPipe;
228                fourPipe = threePipe;
229                threePipe = twoPipe;
230                twoPipe = onePipe;
231                onePipe = inData;
232               }
233               else if (CompTwo) {             // (One, In, Two, Three, Four)
234                fivePipe = fourPipe;
235                fourPipe = threePipe;
236                threePipe = twoPipe;
237                twoPipe = inData;
238               }
239               else if (CompThree) {   // (One, Two, In, Three, Four)
240                fivePipe = fourPipe;
241                fourPipe = threePipe;
242                threePipe = inData;
243               }
244               else if (CompFour) {    // (One, Two, Three, In, Four)
245                fivePipe = fourPipe;
246                fourPipe = inData;
247               }
248               else {                  // (One, Two, Three, Four, In)
249                fivePipe = inData;
250               }
251
252               // Buffer at end of column
253               //if ((i.first && j.first) || (inCol3.address == (ROWS_PER_STRIP-
5))) {
254               if (q_rows.first) {
255                inCol3.address = 0;
256               }
257               tempReg[s] = inData;
258               inData = inCol3;
259               inCol3 = tempReg[s];
260               inCol3.address++;
261             }
262             always if (s==16) {                     // (10)
263             }
264             always if (s==17) {                     // ( 9) +idimp (G)
    (G6)
265               if (gsite) {
266                CompOne = inData > onePipe;
267                CompTwo = inData > twoPipe;
268                CompThree = inData > threePipe;
269                CompFour = inData > fourPipe;
270                CompFive = inData > fivePipe;
```

```
271
272               // Calculate Max
273               if (CompOne) {
274                 maxPipe = inData;
275               }
276               else {
277                 maxPipe = onePipe;
278               }
279
280               // Calculate Min
281               if (CompFive) {
282                 minPipe = fivePipe;
283               }
284               else {
285                 minPipe = inData;
286               }
287
288               // Sort
289               if (CompOne) {                // (X, One, Two, Three, Four)
290                 fivePipe = fourPipe;
291                 fourPipe = threePipe;
292                 threePipe = twoPipe;
293                 twoPipe = onePipe;
294               }
295               else if (CompTwo) {    // (X, In, Two, Three, Four)
296                 fivePipe = fourPipe;
297                 fourPipe = threePipe;
298                 threePipe = twoPipe;
299                 twoPipe = inData;
300               }
301               else if (CompThree) {  // (X, Two, In, Three, Four)
302                 fivePipe = fourPipe;
303                 fourPipe = threePipe;
304                 threePipe = inData;
305               }
306               else if (CompFour) {   // (X, Two, Three, In, Four)
307                 fivePipe = fourPipe;
308                 fourPipe = inData;
309               }
310               else if (CompFive) {   // (X, Two, Three, Four, In)
311                 fivePipe = inData;
312               }
313               else {                 // (X, Two, Three, Four, Five)
314               }
315             }
316           }
317           always if (s==18) {                     // ( 8)
318           }
319           always if (s==19) {                     // ( 7) -idipg (G)
    (G7)
320             if (gsite) {
321               CompMax = inData > maxPipe;
322               CompTwo = inData > twoPipe;
323               CompThree = inData > threePipe;
324               CompFour = inData > fourPipe;
325               CompFive = inData > fivePipe;
326               CompMin = inData > minPipe;
327
328               // Calculate Max
329               if (CompMax) {
330                 maxPipe = inData;
331               }
332
333               // Calculate Min
334               if (!CompMin) {
335                 minPipe = inData;
336               }
337
338               // Sort
339               if (CompTwo) {                // (X, X, Two, Three, Four)
340                 fivePipe = fourPipe;
```

```
341                 fourPipe = threePipe;
342                 threePipe = twoPipe;
343               }
344               else if (CompThree) {  // (X, X, In, Three, Four)
345                 fivePipe = fourPipe;
346                 fourPipe = threePipe;
347                 threePipe = inData;
348               }
349               else if (CompFour) {   // (X, X, Three, In, Four)
350                 fivePipe = fourPipe;
351                 fourPipe = inData;
352               }
353               else if (CompFive) {   // (X, X, Three, Four, In)
354                 fivePipe = inData;
355               }
356               else {                 // (X, X, Three, Four, Five)
357               }
358             }
359           }
360           always if (s==20) {                        // ( 6)
361             //if ((i.first && j.first) || (inCol4.address == (ROWS_PER_STRIP-
5))) {
362             if (q_rows.first) {
363              inCol4.address = 0;
364             }
365             tempReg[s] = inData;
366             inData = inCol4;
367             inCol4 = tempReg[s];
368             inCol4.address++;
369           }
370           always if (s==21) {                        // ( 5) +idim (R/B)
    (RB6)
371             if (!gsite) {
372              CompOne = inData > onePipe;
373              CompTwo = inData > twoPipe;
374              CompThree = inData > threePipe;
375              CompFour = inData > fourPipe;
376              CompFive = inData > fivePipe;
377
378              // Calculate Max
379              if (CompOne) {
380                maxPipe = inData;
381              }
382              else {
383                maxPipe = onePipe;
384              }
385
386              // Calculate Min
387              if (CompFive) {
388                minPipe = fivePipe;
389              }
390              else {
391                minPipe = inData;
392              }
393
394              // Sort
395              if (CompOne) {                // (X, One, Two, Three, Four)
396                fivePipe = fourPipe;
397                fourPipe = threePipe;
398                threePipe = twoPipe;
399                twoPipe = onePipe;
400              }
401              else if (CompTwo) {    // (X, In, Two, Three, Four)
402                fivePipe = fourPipe;
403                fourPipe = threePipe;
404                threePipe = twoPipe;
405                twoPipe = inData;
406              }
407              else if (CompThree) {  // (X, Two, In, Three, Four)
408                fivePipe = fourPipe;
409                fourPipe = threePipe;
```

```
410                 threePipe = inData;
411               }
412               else if (CompFour) {   // (X, Two, Three, In, Four)
413                 fivePipe = fourPipe;
414                 fourPipe = inData;
415               }
416               else if (CompFive) {   // (X, Two, Three, Four, In)
417                 fivePipe = inData;
418               }
419               else {                 // (X, Two, Three, Four, Five)
420               }
421             }
422           }
423           always if (s==22) {                      // ( 4)
424           }
425           always if (s==23) {                      // ( 3) -iinc (G & R/B)
    (G8, RB7)
426             if (gsite) {
427               CompMax = inData > maxPipe;
428               CompThree = inData > threePipe;
429               CompFour = inData > fourPipe;
430               CompFive = inData > fivePipe;
431               CompMin = inData > minPipe;
432
433               // Calculate maximum
434               if (CompMax) {
435                 maxPipe = inData;
436               }
437
438               // Calculate minimum
439               if (!CompMin) {
440                 minPipe = inData;
441               }
442
443               // Sort
444               if (CompThree) {           // (X, X, X, Three, Four)
445                 fivePipe = fourPipe;
446                 fourPipe = threePipe;
447               }
448               else if (CompFour) {       // (X, X, X, inData, Four)
449                 fivePipe = fourPipe;
450                 fourPipe = inData;
451               }
452               else if (CompFive) {       // (X, X, X, Four, InData)
453                 fivePipe = inData;
454               }
455               else {                     // (X, X, X, Four, Five)
456               }
457             }
458             else {                  // RB7
459               CompMax = inData > maxPipe;
460               CompTwo = inData > twoPipe;
461               CompThree = inData > threePipe;
462               CompFour = inData > fourPipe;
463               CompFive = inData > fivePipe;
464               CompMin = inData > minPipe;
465
466               // Calculate Max
467               if (CompMax) {
468                 maxPipe = inData;
469               }
470
471               // Calculate Min
472               if (!CompMin) {
473                 minPipe = inData;
474               }
475
476               if (CompTwo) {               // (X, X, Two, Three, Four)
477                 fivePipe = fourPipe;
478                 fourPipe = threePipe;
479                 threePipe = twoPipe;
```

```
480                }
481                else if (CompThree) {  // (X, X, In, Three, Four)
482                  fivePipe = fourPipe;
483                  fourPipe = threePipe;
484                  threePipe = inData;
485                }
486                else if (CompFour) {   // (X, X, Three, In, Four)
487                  fivePipe = fourPipe;
488                  fourPipe = inData;
489                }
490                else if (CompFive) {   // (X, X, Three, Four, In)
491                  fivePipe = inData;
492                }
493                else {                 // (X, X, Three, Four, Five)
494                }
495              }
496            }
497            if (s==24) {                      // ( 2)
498            }
499            always if (s==25) {                       // ( 1) -idip (G & R/B)
    (RB8)
500              if (!gsite) {
501                CompMax = inData > maxPipe;
502                CompThree = inData > threePipe;
503                CompFour = inData > fourPipe;
504                CompFive = inData > fivePipe;
505                CompMin = inData > minPipe;
506
507                // Calculate maximum
508                if (CompMax) {
509                  maxPipe = inData;
510                }
511
512                // Calculate minimum
513                if (!CompMin) {
514                  minPipe = inData;
515                }
516
517                // Sort
518                if (CompThree) {           // (X, X, X, Three, Four)
519                  fivePipe = fourPipe;
520                  fourPipe = threePipe;
521                }
522                else if (CompFour) {       // (X, X, X, inData, Four)
523                  fivePipe = fourPipe;
524                  fourPipe = inData;
525                }
526                else if (CompFive) {       // (X, X, X, Four, InData)
527                  fivePipe = inData;
528                }
529                else {                     // (X, X, X, Four, Five)
530                }
531              }
532            }
533            always if (s==(STAGES-2)) {
534              CompMax = pixelPipe > maxPipe;
535              CompMin = pixelPipe < minPipe;
536              if (CompMax || CompMin) {
537                outPipe = (fivePipe + fourPipe) >> 1;
538              }
539              else {
540                outPipe = pixelPipe;
541              }
542            }
543          } // Datapath
544        }   // i
545      }     // j
546    }       // k
547
548    // Datapath output thread
549  thread:
```

```
550     for (ko=0; ko<=(IVSIZE-ROWS_PER_STRIP); ko+=(ROWS_PER_STRIP-4)) {
551       wait(outReady);
552       for (jo=0; jo<IHSIZE; jo++) {
553        for (io=0; io<ROWS_PER_STRIP; io++) {
554          Datapath {
555            if (s==STAGES-1) {
556              if ((io>=2) && (io<(ROWS_PER_STRIP-2))) {
557               if (jo<2 || jo>=(IHSIZE-2)) {
558                 outStream = pixelPipe;
559               }
560               else {
561                 outStream = outPipe;
562               }
563              }
564              else if (((io<2) && ko.first) ||
565                       ((io>=(ROWS_PER_STRIP-2)) && ko.last)) {
566               outStream = pixelPipe;
567              }
568            }
569          }
570        }
571       }
572     }
573
574     // Output Stream thread
575   thread:
576     // First Swath (ignore bottom two rows of swath)
577     wait(outReady);
578     for (jsF=0; jsF<IHSIZE; jsF++) {
579       for (isF=0; isF<ROWS_PER_STRIP-2; isF++) {
580        Datapath {
581          if (s==STAGES-1) {
582            out[isF][jsF] = outStream;
583          }
584        }
585       }
586       Datapath {}
587       Datapath {}
588     }
589
590     // Middle Swaths (ignore top and bottom two rows of swath)
591     for (ksM=(ROWS_PER_STRIP-4); ksM<(IVSIZE-ROWS_PER_STRIP);
ksM+=(ROWS_PER_STRIP-4)) {
592       wait(outReady);
593       for (jsM=0; jsM<IHSIZE; jsM++) {
594        Datapath {}
595        Datapath {}
596        for (isM=2; isM<(ROWS_PER_STRIP-2); isM++) {
597          Datapath {
598            if (s==STAGES-1) {
599              out[ksM+isM][jsM] = outStream;
600            }
601          }
602        }
603        Datapath {}
604        Datapath {}
605       }
606     }
607
608     // Last Swath (ignore top two rows of swath)
609     wait(outReady);
610     for (jsL=0; jsL<IHSIZE; jsL++) {
611       Datapath {}
612       Datapath {}
613       for (isL=2; isL<ROWS_PER_STRIP; isL++) {
614        Datapath {
615          if (s==STAGES-1) {
616            out[isL+(IVSIZE-ROWS_PER_STRIP)][jsL] = outStream;
617          }
618        }
619       }
```

```
620      }
621    }     // Par
622  }       // med_filt_rapid
623
```

Driver Program: med_filtTest.cc

```
     1 ////////////////////////////////////////////////////////////////////
     2 //
     3 //                  Driver Program for
     4 //                   Median Filter
     5 //
     6 // Kevin Rennie
     7 // 21 March 2000
     8 //
     9 // med_filtTest.cc
    10 //
    11 ////////////////////////////////////////////////////////////////////
    12
    13 #include <assert.h>
    14 #include <stdio.h>
    15 #include <iostream.h>
    16 #include <fstream.h>
    17 #include <iomanip.h>
    18 #include <math.h>
    19 #include <stdlib.h>
    20 #include <malloc.h>
    21 #include "med_filt.h"
    22
    23 // Data structures implementation of med_filt
    24 typedef unsigned char byte;
    25 typedef byte boolean;
    26 typedef struct {
    27   int *s;
    28   int *i;
    29   int *j;
    30 } DEFMAP;
    31 typedef struct {
    32   short int *sm;
    33   short int *us;
    34 } VTMP;
    35
    36 extern void med_filt_rapid(Word in[IVSIZE][IHSIZE], Word out[IVSIZE][IHSIZE]);
    37 /*
    38 extern void med_filt_rapid(Word in[IVSIZE][IHSIZE],
    39                          Word outA[IVSIZE][IHSIZE], Word outB[IVSIZE][IHSIZE], Word
outC[IVSIZE][IHSIZE]);
    40                          */
    41
    42 ////////////////////////////////////////////////////////////////////
    43 //
    44 //  Test Utility Functions
    45 //
    46 ////////////////////////////////////////////////////////////////////
    47
    48 // prints a single matrix (ie. R, G, or B)
    49 void print_matrix(Word img[IVSIZE][IHSIZE]) {
    50   int i, j;
    51
    52   for (i = 0; i < IVSIZE; i++) {
    53     for (j = 0; j < IHSIZE; j++) {
    54       printf("%4d ", img[i][j]);
    55     }
    56     printf("\n");
    57   }
    58 }
    59
    60
    61 void print_data(Word A[IVSIZE][IHSIZE], Word B[IVSIZE][IHSIZE], Word
C[IVSIZE][IHSIZE])
    62 {
    63   printf("\nA:\n");
    64   print_matrix(A);
    65   printf("\nB:\n");
    66   print_matrix(B);
```

```
 67   printf("\nC:\n");
 68   print_matrix(C);
 69 }
 70
 71 void print_vector(short int *img)
 72 {
 73   int i,j,k;
 74   int iptr = 0;
 75   int increment = 1;
 76
 77   for (i = 0; i < IVSIZE; i++) {
 78     for (j = 0; j < IHSIZE; j++) {
 79       printf("%4d ", img[iptr]);
 80       iptr += increment;
 81     }
 82     printf("\n");
 83   }
 84 }
 85
 86 void test_data(Word out[IVSIZE][IHSIZE], short int *test_vector) {
 87   int i,j;
 88   int iptr = 0;
 89   int print_errors = 1;
 90   int print_difference = 1;
 91   int correct = TRUE;
 92   int increment = 1;
 93
 94   Word Diff[IVSIZE][IHSIZE];
 95
 96   printf("\n\nTesting results...");
 97   if (print_errors)
 98     printf("\n\nErrors:");
 99   for (i=0; i<IVSIZE; i++) {
100     for (j=0; j<IHSIZE; j++) {
101       Diff[i][j] = out[i][j] - test_vector[iptr];
102       if (Diff[i][j] != 0) {
103         if (print_errors)
104           printf("\nError occured at row %d, col %d, test = %d, rapid = %d",
105                  i, j, test_vector[iptr], out[i][j]);
106         correct = FALSE;
107       }
108       iptr+=increment;
109     }
110   }
111
112   if (correct == FALSE) {
113     if (print_difference) {
114       printf("\n\nHere's the diffrence between Rapid and testbench:\n\n");
115       print_matrix(Diff);
116     }
117     printf("\n\n----> Verification Failed! <-----\n\n");
118   }
119   else
120     printf("\n\n----> Verification Succeeded! (woohoo) <-----\n\n");
121 }
122
123 short int *med_filt(short int *img, int rank, int ihsize, int ivsize)
124 {
125   //FILE *fout;
126   register int i, j, x, y, p, iptr, optr, ijmp, idim, idip, iinc, ohsize, ovsize;
127   register int cptr, idimg, idipg, xhold;
128   int hpix, cpix, scy_shift, med_shift, sev_shift;
129   short int hival, loval, pixval, tmp, *tz, *rep;
130   short int by00, by01, by02, by03, by04, by05, by06, by07;
131   short int by10, by11, by12, by13, by14, by15, by16, by17;
132   short int by20, by21, by22, by23, by24, by25, by26, by27;
133   short int by30, by31, by32, by33, by34, by35, by36, by37;
134   short int by40, by41, by42, by43, by44, by45, by46, by47;
135   short int by50, by51, by52, by53, by54, by55, by56, by57;
136   short int by60, by61, by62, by63, by64, by65, by66, by67;
137   short int by70, by71, by72, by73, by74, by75, by76, by77;
```

```
   short int by81;
   short int by90;
   short int def_thresh_used, severity;
   char c, filestring[80];
   char whoami[64] = "RaPiDTest";             // Added for RaPiD driver program
   DEFMAP map;
   boolean chuckit = FALSE;

   ohsize = ihsize - 4;
   ovsize = ivsize - 4;

   map.s = (int *)malloc(sizeof(int)*(DEF_MAXDEFS+1));
   map.i = (int *)malloc(sizeof(int)*(DEF_MAXDEFS+1));
   map.j = (int *)malloc(sizeof(int)*(DEF_MAXDEFS+1));
   rep = (short int *)malloc(sizeof(short int)*ihsize*ivsize*COLORS);
   if(DEF_REPORT)
     for(i=0; i<ihsize*ivsize*COLORS; i++)
       rep[i] = 0;

   def_thresh_used = (DEF_THRESH << (DE_B3 - DE_B4));
   scy_shift = DE_B0 - DE_B1;
   med_shift = DE_B0 - DE_B2;
   sev_shift = DE_B0 - DE_B3;
   /* scy_shift = 0; */

   tz = (short int *)malloc(sizeof(short int)*ihsize*ivsize);
   /* copy in image to tmp to ease coding */
   for(i=0; i<ihsize*ivsize; i++)
     tz[i] = img[i];
   ijmp = ihsize << 1;
   iinc = 2;
   idim = ijmp - iinc;
   idip = ijmp + iinc;
   idimg = idim >> 1;
   idipg = idip >> 1;
   iptr = idip;
   cptr = COLORS * idip;
   for(p=0; p<DEF_MAXDEFS+1; p++) map.s[p] = -(p+50);
   optr = 0;
   /* scythe filter detect, correct if requested */
   for(i=0; i<ovsize; i++) {
     for(j=0; j<ohsize; j++) {
       /* original or recursive neighbourhood assignments, centre included, tighter
on green checkers */
       pixval = img[iptr];
       if(((i & 1) ^ VJOG) == ((j & 1) ^ HJOG)) {
         hpix = sev_shift;
         cpix = GRN;
         by00 = DEF_ORIGIN ? img[iptr - idipg] : tz[iptr - idipg];
         by01 = DEF_ORIGIN ? img[iptr - ijmp] : tz[iptr - ijmp];
         by02 = DEF_ORIGIN ? img[iptr - idimg] : tz[iptr - idimg];
         by03 = DEF_ORIGIN ? img[iptr - iinc] : tz[iptr - iinc];
         by04 = img[iptr + iinc];
         by05 = img[iptr + idimg];
         by06 = img[iptr + ijmp];
         by07 = img[iptr + idipg];
       }
       else {
         hpix = sev_shift + 1;
         cpix = ((i & 1) ^ VJOG) ? BLU : RED;
         by00 = DEF_ORIGIN ? img[iptr - idip] : tz[iptr - idip];
         by01 = DEF_ORIGIN ? img[iptr - ijmp] : tz[iptr - ijmp];
         by02 = DEF_ORIGIN ? img[iptr - idim] : tz[iptr - idim];
         by03 = DEF_ORIGIN ? img[iptr - iinc] : tz[iptr - iinc];
         by04 = img[iptr + iinc];
         by05 = img[iptr + idim];
         by06 = img[iptr + ijmp];
         by07 = img[iptr + idip];
       }
       /* batcher-banyan sort on 8-ring pels */
       if((by00 >> scy_shift) > (by01 >> scy_shift)) { by10 = by00; by11 = by01; }
```

```
   208        else
{ by10 = by01; by11 = by00; }
   209        if((by02 >> scy_shift) < (by03 >> scy_shift)) { by12 = by02; by13 = by03; }
   210        else
{ by12 = by03; by13 = by02; }
   211        if((by04 >> scy_shift) > (by05 >> scy_shift)) { by14 = by04; by15 = by05; }
   212        else
{ by14 = by05; by15 = by04; }
   213        if((by06 >> scy_shift) < (by07 >> scy_shift)) { by16 = by06; by17 = by07; }
   214        else
{ by16 = by07; by17 = by06; }
   215        if((by10 >> scy_shift) > (by12 >> scy_shift)) { by20 = by10; by21 = by12; }
   216        else
{ by20 = by12; by21 = by10; }
   217        if((by11 >> scy_shift) > (by13 >> scy_shift)) { by22 = by11; by23 = by13; }
   218        else
{ by22 = by13; by23 = by11; }
   219        if((by14 >> scy_shift) < (by16 >> scy_shift)) { by24 = by14; by25 = by16; }
   220        else
{ by24 = by16; by25 = by14; }
   221        if((by15 >> scy_shift) < (by17 >> scy_shift)) { by26 = by15; by27 = by17; }
   222        else
{ by26 = by17; by27 = by15; }
   223        if((by20 >> scy_shift) > (by22 >> scy_shift)) { by30 = by20; by31 = by22; }
   224        else
{ by30 = by22; by31 = by20; }
   225        if((by21 >> scy_shift) > (by23 >> scy_shift)) { by32 = by21; by33 = by23; }
   226        else
{ by32 = by23; by33 = by21; }
   227        if((by24 >> scy_shift) < (by26 >> scy_shift)) { by34 = by24; by35 = by26; }
   228        else
{ by34 = by26; by35 = by24; }
   229        if((by25 >> scy_shift) < (by27 >> scy_shift)) { by36 = by25; by37 = by27; }
   230        else
{ by36 = by27; by37 = by25; }
   231        if((by30 >> scy_shift) > (by34 >> scy_shift)) { by40 = by30; by41 = by34; }
   232        else
{ by40 = by34; by41 = by30; }
   233        if((by31 >> scy_shift) > (by35 >> scy_shift)) { by42 = by31; by43 = by35; }
   234        else
{ by42 = by35; by43 = by31; }
   235        if((by32 >> scy_shift) > (by36 >> scy_shift)) { by44 = by32; by45 = by36; }
   236        else
{ by44 = by36; by45 = by32; }
   237        if((by33 >> scy_shift) > (by37 >> scy_shift)) { by46 = by33; by47 = by37; }
   238        else
{ by46 = by37; by47 = by33; }
   239        if((by40 >> scy_shift) > (by44 >> scy_shift)) { by50 = by40; by51 = by44; }
   240        else
{ by50 = by44; by51 = by40; }
   241        if((by42 >> scy_shift) > (by46 >> scy_shift)) { by52 = by42; by53 = by46; }
   242        else
{ by52 = by46; by53 = by42; }
   243        if((by41 >> scy_shift) > (by45 >> scy_shift)) { by54 = by41; by55 = by45; }
   244        else
{ by54 = by45; by55 = by41; }
   245        if((by43 >> scy_shift) > (by47 >> scy_shift)) { by56 = by43; by57 = by47; }
   246        else
{ by56 = by47; by57 = by43; }
   247        if((by50 >> scy_shift) > (by52 >> scy_shift)) { by60 = by50; by61 = by52; }
   248        else
{ by60 = by52; by61 = by50; }
   249        if((by51 >> scy_shift) > (by53 >> scy_shift)) { by62 = by51; by63 = by53; }
   250        else
{ by62 = by53; by63 = by51; }
   251        if((by54 >> scy_shift) > (by56 >> scy_shift)) { by64 = by54; by65 = by56; }
   252        else
{ by64 = by56; by65 = by54; }
   253        if((by55 >> scy_shift) > (by57 >> scy_shift)) { by66 = by55; by67 = by57; }
   254        else
{ by66 = by57; by67 = by55; }
```

```
   255
   256       /* variable-rank scythe filter */
   257       switch(rank) {
   258       case 1: hival = by60; loval = by67; break;
   259       case 2: hival = by61; loval = by66; break;
   260       case 3: hival = by62; loval = by65; break;
   261       default: hival = by63; loval = by64;
   262       }
   263       if((pixval >> scy_shift) > (hival >> scy_shift)) {
   264         tz[iptr] = (by63+by64) >> 1;
   265         printf("\ntest: Sorted   pixval %2d, list: %2d, %2d, %2d, %2d, %2d, %2d,
%2d, %2d",pixval,by60,by61,by62,by63,by64,by65,by66,by67);
   266         printf("\ntest: Original pixval %2d, list: %2d, %2d, %2d, %2d, %2d, %2d,
%2d, %2d\n",pixval,by00,by01,by02,by03,by04,by05,by06,by07);
   267         //tz[iptr] = hival;
   268       }
   269       else if((pixval >> scy_shift) < (loval >> scy_shift)) {
   270         tz[iptr] = (by63+by64) >> 1;
   271         printf("\ntest: Sorted   pixval %2d, list: %2d, %2d, %2d, %2d, %2d, %2d,
%2d, %2d",pixval,by60,by61,by62,by63,by64,by65,by66,by67);
   272         printf("\ntest: Original pixval %2d, list: %2d, %2d, %2d, %2d, %2d, %2d,
%2d, %2d\n",pixval,by00,by01,by02,by03,by04,by05,by06,by07);
   273         //tz[iptr] = loval;
   274       }
   275       else
   276         tz[iptr] = pixval;
   277       severity = abs(tz[iptr] - pixval) >> hpix;
   278       if(DEF_SCYTHE)
   279         rep[COLORS * iptr + cpix] = abs(tz[iptr] - pixval) >> hpix;
   280       /* insert in map if appropriate */
   281       if(severity < def_thresh_used) severity = 0;
   282       p = DEF_MAXDEFS;
   283       while((--p) >= 0) {
   284         if(map.s[p] < severity) {
   285           map.s[p+1] = map.s[p];
   286           map.i[p+1] = map.i[p];
   287           map.j[p+1] = map.j[p];
   288           map.s[p] = severity;
   289           map.i[p] = i + 2;
   290           map.j[p] = j + 2;
   291         }
   292         else {
   293           break;
   294         }
   295       }
   296       if(!DEF_SCYTHE)
   297         tz[iptr] = pixval;
   298       ++iptr;
   299       cptr += COLORS;
   300       ++optr;
   301     }
   302     cptr += 4 * COLORS;
   303     iptr+=4;
   304   }
   305
   306   free(rep);
   307   free(map.s); free(map.i); free(map.j);
   308   return(tz);
   309
   310 }
   311
   312 ////////////////////////////////////////////////////////////////////
   313 //
   314 //  Main Function
   315 //
   316 ////////////////////////////////////////////////////////////////////
   317
   318 int main() {
   319   int i, j, k;
   320   int iptr, tmp;
   321
```

```
   322    int data_seq = 1;
   323    int data_rand = 2;
   324    int data_type = data_rand;
   325    int correct = TRUE;
   326
   327    srandom(22);
   328
   329    Word in[IVSIZE][IHSIZE];            // Input matrix
   330    Word in_v[IVSIZE*IHSIZE];                  // Input vector
   331    Word out[IVSIZE][IHSIZE];           // Output matrix
   332    Word out_v[IVSIZE*IHSIZE];          // Output vector
   333    Word cons_vec[CONSTANTS];           // Constants for matrix calculation
   334    Word outA[IVSIZE][IHSIZE];          // Output matrix
   335    Word outB[IVSIZE][IHSIZE];          // Output matrix
   336    Word outC[IVSIZE][IHSIZE];          // Output matrix
   337
   338    int scy_shift, med_shift, sev_shift;
   339
   340    short int *ip, *de;                 // testbench matrices, ip input to med_filt,
de output from med_filt
   341
   342    ip = (short int *)malloc(sizeof(short int)*IHSIZE*IVSIZE);
   343
   344    // Generate input data
   345    for (j = 0; j < IHSIZE; j++) {
   346      for (i = 0; i < IVSIZE; i++) {
   347        if (data_type == data_seq) {
   348          tmp = j*IVSIZE + i;           // column major order
   349          in[i][j] = tmp;
   350        }
   351        else if (data_type == data_rand) {
   352          in[i][j] = RAND(63) + 1;
   353        }
   354      }
   355    }
   356
   357    // Generate input vector from input matrix
   358    iptr = 0;
   359    for (i = 0; i < IVSIZE; i++) {
   360      for (j = 0; j < IHSIZE; j++) {
   361        ip[iptr++] = in[i][j];
   362      }
   363    }
   364
   365    for (i = 0; i < IHSIZE*IVSIZE; i++) {
   366      in_v[i] = ip[i];
   367    }
   368
   369    /*
   370    // Build constant vector
   371    printf("\nConstants:\n  ");
   372    for (i=0; i<CONSTANTS; i++) {
   373      printf("%d = %d, ", i, cons_vec[i]);
   374    }
   375    printf("\n");
   376    */
   377
   378    // Defect Correction
   379    de = med_filt(ip, 1, IHSIZE, IVSIZE);
   380
   381    // Rapid Calculation
   382    printf("\nInput Data:\n==========\n");
   383    print_matrix(in);
   384    med_filt_rapid(in, out);
   385    //med_filt_rapid(in, outA, outB, outC);
   386    printf("\nOutput Data:\n===========\n");
   387    print_matrix(out);
   388
   389    printf("\n\nMed_Filt Data:\n===========\n");
   390    print_vector(de);
   391
```

```
   392    /*
   393    printf("\n");
   394    iptr = 0;
   395    for (i = 0; i < IVSIZE; i++) {
   396      for (j = 0; j < IHSIZE; j++) {
   397        if ((in[i][j] != de[iptr]) || ((outB[i][j] != de[iptr]) && (outB[i][j]>0)))
   398          printf("\n(%2d,%2d) - in: %2d, rpd: %2d, test: %2d, max: %2d, min:
%2d",i,j,in[i][j],outB[i][j],de[iptr],outA[i][j],outC[i][j]);
   399        iptr++;
   400      }
   401    }
   402    */
   403
   404    // Compare rapid computation and testbench
   405    test_data(out, de);
   406
   407
   408    free(ip);
   409    return 0;
   410  }
```