Smart Camera Based on Reconfigurable Hardware Enables Diverse Real-time Applications

Miriam Leeser, Shawn Miller and Haiqian Yu
Northeastern University
Boston, MA 02115
mel@ece.neu.edu, smiller@ece.neu.edu, hyu@ece.neu.edu

Abstract

We demonstrate the use of a "smart camera" to accelerate two very different image processing applications. The smart camera consists of a high quality video camera and frame grabber connected directly to an FPGA processing board. The advantages of this setup include minimizing the movement of large datasets and minimizing the latency by starting to process data before a complete frame has been acquired. The two applications, one from the area of medical image processing and the other from computational fluid dynamics both exhibit speedups of more than 20 times over software implementations on a 1.5GHz PC. This smart camera setup is enabling image processing implementations that have not, before now, been achievable in real time.

1. Introduction

We demonstrate the use of a single hardware platform to accelerate two very different applications, one from the area of medical image processing and the other from computational fluid dynamics. Both applications involve implementations that process raw pixel data directly from a camera. Both can begin processing image data before a complete frame of data has been acquired. In both cases, real-time implementations have been desirable but not achievable up to now.

The medical image processing application is RVT: Retinal Vascular Tracing. An algorithm for tracing vasculature in retinal images has been developed by our colleagues at RPI [2]. However its software implementation is too slow for real-time operation. By implementing the computationally complex components in reconfigurable hardware, we enable the real-time implementation of RVT. This will allow surgeons to see live retinal images with vasculature highlighted in real time during surgery. An important requirement of this system is to minimize the total latency so that the image is displayed with very little overall delay.

The computational fluid dynamics application is PIV: Particle Image Velocimetry. Real-time PIV will enable applications such as active control flap control for airfoils, resulting in more efficient aerodynamics. Currently, this control is done based on precomputed data rather than on information of the current flow of gasses over the airfoil.

Both these applications directly process images acquired from a camera. In both cases, low overall latency is essential, and a large amount of data must be acquired and processed. Our smart camera setup feeds raw data directly to an FPGA accelerator board. The FPGA accelerator board can start processing image data before an entire frame has been captured, thus minimizing latency. By feeding the data directly from camera to FPGA board, data movement is minimized and slow downs in processing due to accessing large datasets through the memory subsystem of a workstation are avoided. The acceleration in both of these applications is due to the closeness of the FPGA hardware to the sensor (the camera) as well as the suitability of the specific applications to reconfigurable hardware acceleration.

The idea of using FPGA technology to create a smart camera is not a new one. In 1991, Sartori [11] designed an FPGA board that captured data directly from an image sensor, implemented an edge detection algorithm, and facilitated a parallel data interface to the host processor. Although the input images only had a resolution of 32 x 32 pixels with 1-bit monochrome pixel data, and the FPGA was running between 3 and 4 MHz, the idea of moving image processing to hardware and closer to the sensor enabled a real-time solution for his application. Scalera, et al. [12] designed the CAμS board which included an FPGA and a DSP that interfaced directly to microsensors and preprocessed the data in order to reduce the amount of information necessary to transmit to the host processor. The goals of that project were to minimize power consumption and size. Lienhart, et al. [8] were able to implement real-time video compression on three image sequences in parallel at

Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)
0-7695-2230-0/04 $ 20.00 IEEE

30 fps using FPGAs. This design has a high latency because an entire frame was stored in memory before the compression algorithm began. Their input data came from the host PC via the PCI bus; however it was noted that the solution could be implemented with a direct camera connection. Our approach differs from earlier approaches in the quality of the camera, speed of processing, and the minimization of latency by processing data straight from the camera.

The rest of this paper is organized as follows. First we describe the common smart camera hardware setup in more detail. Next we describe the details of the two applications, Retinal Vascular Tracing (RVT) and Particle Image Velocimetry (PIV). These sections include details of the implementations and performance results. Finally, we present conclusions and future directions.

2. Hardware Setup

Our hardware setup includes a high quality video camera, a framegrabber, and an FPGA board. Figure 1 shows how these parts are connected. The camera is a Dalsa 1M30P which outputs 12-bit data at variable rates and image resolutions. Data rates are up to 30 frames per second. Images can be 1024x1024 or 512x512 pixels. The camera outputs raw pixel data at frame rate. The camera's data cable connects to a framegrabber designed by Dillon Engineering. The FPGA board is the Firebird from Annapolis Microsystems, which has a Xilinx Virtex 2000E and five on-board memory chips. The framegrabber is a daughter card to the Firebird, and interfaces through an I/O port which sends data directly to the FPGA. The FPGA interfaces to the host PC through the PCI bus, which is 64-bits wide running at 66 MHz.

The FPGA design consists of two parts running concurrently. One part takes the 12-bit pixel data from the framegrabber and packs it into 64-bit words that is then stored in the on-board memory. The data is written to, and later read from, one of two memory chips. The Memory Switching Design coordinates which chip to use, what address to access, and performs all of the memory handshaking operations. When the data is read, it is stored in the on-chip memory, or BlockRAM, for faster access. The second part, the Image Processing Design, is a custom block, which contains the logic for application specific algorithms. The results from the Image Processing Design are then stored in one of two output memory chips. The host PC makes calls to the PCI bus to access the results from the Firebird memory.

Note that we block up and store data from the camera in on-chip memory before processing it on the FPGA. An alternative to this approach is to process the streaming data directly. We do not do this because the image processing algorithms we implement are window based operations with large windows. For RVT, the result for one pixel requires an 11x11 window. For PIV, the window size depends on the application. Due to the window-based nature of the algorithms, we chose to separate forming windows from processing windows. Note that the image processing design can begin processing windows as soon as sufficient data is stored in memory. It does not need to wait for a frame to be acquired before processing is started.

Our overall hardware approach allows us to reuse much of our design across several different applications. The data packing and memory switching designs remain the same for the two implementations described in this paper.

3. Retinal Vascular Tracing

Real-time tracing of the vascular structure of retinal images is an important part of several medical applications, including live laser surgery. Some unique challenges are involved because of the need to attain reliable results from images that are acquired in non-ideal conditions. Poor illumination in conjunction with the inability to completely stop all body and eye movements in a live setting contribute to the challenges. These problems are further complicated by the fact that high data rates require a large amount of computation to be performed in very short periods of time.

With real-time vascular tracing results, it is possible for a surgeon to have an image of the retina that is being operated on with highlighted blood vessels. It is also possible to create a system where the surgical laser can be automatically shut off if it is detected that it is off-course. For these applications to be possible, not only do the results have to be computed at the same rate that the image data is output from the camera, but it is also important that there is very little delay in making the results available.

3.1. RVT Algorithm

The Retinal Vascular Tracing (RVT) Algorithm was designed by researchers at Rensselaer Polytechnic Institute. The core of the tracing algorithm is a two-dimensional convolution with 16 independent filters, described in Can, et al. [2]. These are directional matched filters where each filter corresponds to a unique direction, separated by 22.5°, as shown in Figure 2. Each filter response is calculated, and the filter that returns the greatest response is used in the tracing algorithm to determine which direction to continue looking for the next point on the vessel. The RVT algorithm includes other computations besides the filter response calculations, but this is the most computationally intensive part.

Analyzing the filter response calculations reveals some optimizations to reduce the computational complexity.

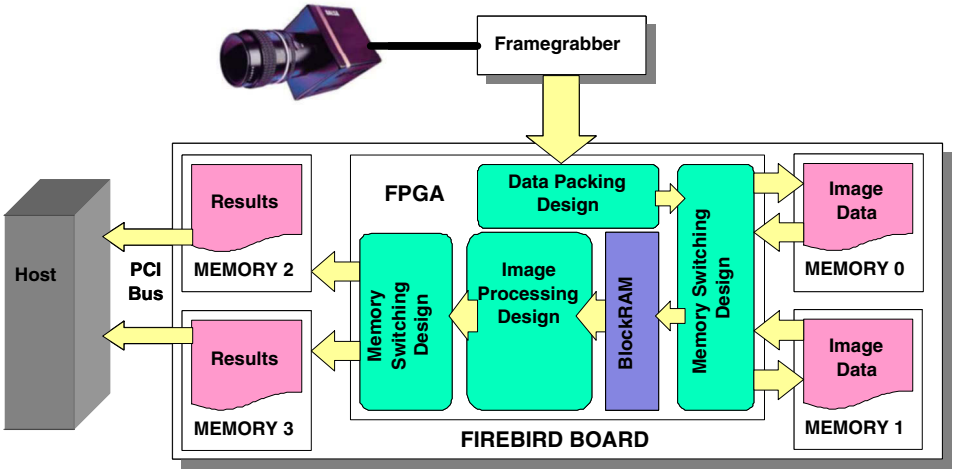

Figure 1. Hardware Setup

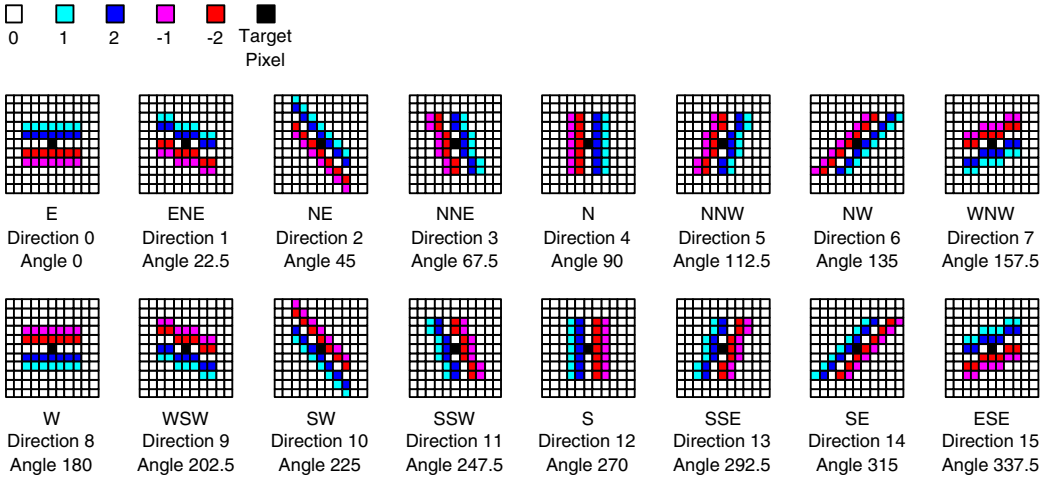

Figure 2. Matched Filters

First, all of the filter coefficients are ± 1 and ± 2. The corresponding multiplications can be replaced with simpler binary shifts and sign changes. It can also be noted from Figure 2 that the filters in the first row are the same as the filters in the second row, except for a change in sign. Only eight of the responses need to be calculated; the other eight can be found by simply copying the results and changing the sign. All of the template responses are also independent of each other. This implies that given one 11 x 11 pixel block of data, all of the template responses can be calculated in parallel, and the results returned at the same time.

3.2. Hardware System Design

From the algorithm analysis, it is shown that the filter response calculations can be computed as a series of parallel shifts and adds. A hardware solution can take advantage of the parallelism, and offer a great deal of speedup compared to a serial solution implemented in software.

The "Smart Camera" in Figure 1 is implemented by implementing the filter response calculations as the Image Processing Design block of the FPGA. Instead of the host processor getting image data directly from the camera, it receives both the unaltered image data and the filter response results for every pixel from the hardware board. This additional data is received at the same rate as the original camera data would be, with a small delay to allow for the additional calculation.

By moving the most computationally complex part of the RVT algorithm into hardware and closer to the camera, the amount of software computation is greatly reduced. The overall processing time is reduced to create a high-speed

run-time solution.

3.2.1 Filter Response Calculation

Figure 3 shows how the filter responses are calculated on the FPGA. The data for the 11 x 11 neighborhood of pixels that

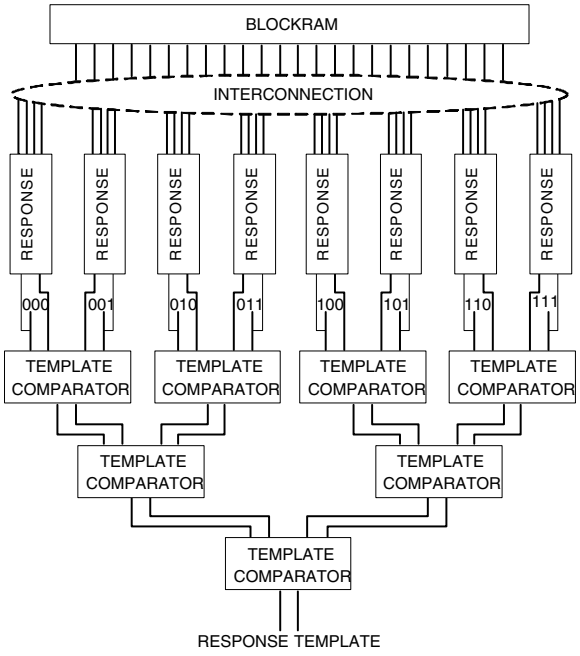

Figure 3. Filter Response Hardware Implementation

is required for the filters is stored in the on-chip memory, or BlockRAM. There are eight identical copies of the response module with four inputs, one for each coefficient (± 1 and ± 2). The interconnection network chooses the correct data to send to each input of the response modules. The response modules perform the binary shifts for the data with coefficients of ± 2, then compute the response with additions and subtractions. The response for the filter's complement is found by changing the sign of the response that was calculated. Since we are only interested in finding the response with the greatest value, we output the absolute value of the response along with an additional bit showing whether the result represents the filter or its complement. Cerro-Prada and James-Roxby describe methods for implementing 3 x 3 convolutions with arbitrary weights using distributed arithmetic [3]. Our weights are sufficiently simple that such an approach is not needed here.

The next step is to compare the eight responses that were just calculated to find the greatest result. This is accomplished in three steps using a comparator tree. The response and 4-bit label of a pair of filters are input into a template comparator module, and the output is the response and label of the filter with the greatest result. After all of the comparisons are complete, the filter with the greatest response is output, and ready to be stored into the output memory for the host to use.

The pipeline registers are not shown in Figure 3. In order to keep a fast clock speed, there is a pipeline register following every addition, subtraction, and comparison. While these registers require additional space on the FPGA and add several clock cycles of latency, they are necessary so that we can achieve clock speeds that can keep up with the frame rate of the camera.

3.2.2 Traversing the Image

The data coming from the camera is 12-bits per pixel. In order to efficiently store the pixel data into our 64-bit wide memory and minimize the number of memory reads and writes, we have a data packing design that packs five pixels into a single word of memory. A similar data packing technique was used in [6]. The filter response calculations require an 11 x 11 pixel window, but since we get five pixels from a single memory read, we cannot efficiently input a window that is 11 pixels wide. Instead, we read in windows that are 11 rows by 15 columns wide, and find five results for every window that is read.

Adjacent pixel windows contain a great deal of common data, so rereading an entire window before processing would require more memory accesses than is necessary. To avoid this, we have a system for shifting and reusing a majority of the data without accessing the memory again. The 11 x 15 pixel window is stored in BlockRAM which is split into three sections. Each section stores an 11 x 5 portion of the window, which is a single column of 11 words from memory. When the results for a window are finished, the window is shifted to the right by five pixels for the next set of calculations. Figure 4 shows that in order to get a new window, only one new 11 x 5 block of pixels needs to be read. The other two 11 x 5 blocks are still in the BlockRAM from the previous window. Thus, we need to perform 11 memory reads in order to get five results. Different methods for buffering image data are described in [7].

After the window gets to the end of one row, it needs to shift down one pixel and start at the beginning of the next row. Since the data is stored in the memory in raster scan order, if we continue to increment the addresses in the same manner, the window will automatically shift down to the next line as desired. At the end of every row, however, there are two filter calculations that are invalid because the window spans two different rows. This is a price that we are willing to pay, because hardware algorithms are more efficient when there are fewer special cases, and can run uninterrupted over large sets of data.

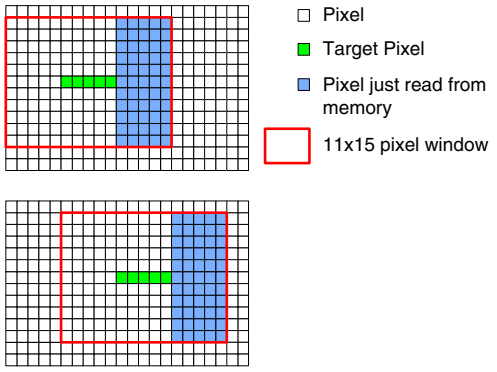

Figure 4. Shifting the Window

The frames are 512 x 512 pixels, so we pad the right hand side of the image with three columns of zeros to create a 512 x 515 image. This allows us to continue shifting the window to the right by five pixels without creating a problem at the end of a row. Due to the nature of the application, boundary conditions are not an issue. There is no important information on the edges of the image, so any invalid results that occur when an 11 x 11 window spans multiple rows can simply be ignored.

3.2.3 Memory Interface

Memory management is an important consideration in real-time hardware solutions. Data streams in from the framegrabber continuously, so the incoming data must be stored and overwritten carefully. Another, larger problem, is that a single on-board memory chip cannot be read from and written to on the same clock cycle. Like many other hardware applications, reading data from off-chip memory into the FPGA is the bottleneck in speeding up the design. The only way to prevent the entire design from having to wait for new data, is to read a new word from memory on every clock cycle. However, if every clock cycle is dedicated to reading data, then there would be no free clock cycles to write new data to the same memory chip. To solve this problem, two memory chips must be used to store the incoming data.

One solution to the problem is to store the first frame that comes from the camera into Memory 0. The next frame is stored into Memory 1, and while new data is being written into Memory 1, the data from Memory 0 could be read into the FPGA with no conflicts. This "ping-pong" method would continue, so the memory chip that is being read from and written to is alternating every frame, avoiding concurrent reads and writes on the same chip. While this is a common solution to streaming data applications, it introduces unacceptable delay in our application. No processing can be done until the first frame is completely written, and this latency carries through the entire design.

In order to calculate all of the filter responses shown in Figure 2, an 11 x 11 pixel neighborhood is required. The data coming from the camera is in raster order, so we can theoretically start processing data after the first 11 rows of data are available. In order to make this happen, we designed a more complex memory swapping method. After the Data Packing Design packs five incoming pixels into a 64-bit word, every consecutive word is stored in two alternating memories. The first word into Memory 0, the second word into Memory 1, etc. After the first 11 rows of data are written, the FPGA can read the data in the correct order by alternating reads between the two memories. Using this method, data is still being read on every clock cycle so that the processing doesn't slow down. In addition, both memory chips are only being accessed on every other clock cycle for a read, and new data can be written on the off cycles. Figure 5 shows a basic timing diagram of how the reads and writes alternate. Note that there are more reads than writes in this diagram. Due to the nature of reading two-dimensional blocks of data, we must re-read a great deal of data, so there is not a one-to-one relationship between reads and writes.

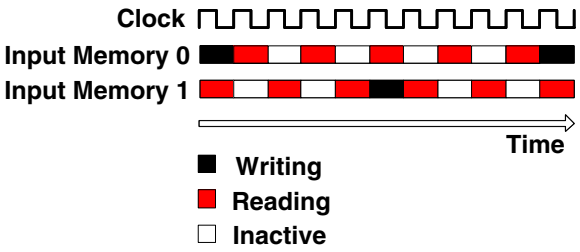

Figure 5. Memory Timing Diagram

3.3. Results and Analysis

The goal of the design is for the processing to keep up with the frame rate of the camera. To achieve this, the design was synthesized and run at 60 MHz. At this speed, the design is able to process the incoming image data, and output it at the frame rate of 30 frames per second (fps). The biggest concern is minimizing the latency. Table 1 shows the latency for each part of the hardware design.

The total latency from the time the camera sends the first pixel to the time that the first filter response is available in the on-board memory is approximately 250μsec. The majority of the latency comes from waiting for the initial data to become available in on-board memory. The framegrabber latency is unavoidable, because one line of pixels is internally cached. The latency for computation and the storing of results is negligible due to the large amount of pipelining in the design.

Operation	*Clock Cycles*	*Clock Speed*	*Latency*
Framegrabber	∼520	30 MHz	17μsec
Write 11 rows	∼7000	30 MHz	235μsec
Compute Results	∼20	60 MHz	0.3μsec
Write Results	∼5	60 MHz	0.08μsec

Table 1. Latency

A 250μsec latency is a very small price to pay, especially when considering that at 30 fps, it takes the camera 33msec just to output one frame. We took the filter response algorithm that was implemented in hardware, and wrote it in C to compare the timing results. Random image data was written to memory, and we found the time that it took to read an 11 x 11 window from memory, run the eight unique filters over the window, and decide which filter returned the greatest response. The program was run 50 million times on an Intel Xeon 2.6GHz PC, and we found that it took, on average, 1.96μsec to find the result for one window. The hardware solution is able to load five new windows in 11 clock cycles. When the pipeline is full, the FPGA can return five results every 11 clock cycles. Running at 60MHz, one result, on average, is returned every 36.67nsec. The speedup factor in hardware is greater than 50 compared to our software implementation.

4. Particle Image Velocimetry

Particle Image Velocimetry(PIV) [10] is a measuring technique to capture the flow velocity of fluids. For a conventional PIV system, small particles are added to the fluid and their movements are measured by comparing two images taken within a short time interval, t and $t+\triangle t$, of the flow field. Such PIV systems are called double frame/single exposure systems and the movement of the fluid can be estimated by comparing these two images recorded in the two frames.

A typical PIV system is shown in Figure 6. It is composed of the follows parts: camera, frame grabber, image processing unit and the output. The camera is the PIV recording part. It can be either $single\ frame/multi\text{-}exposure$ or $multi\text{-}frame/single\ exposure$ PIV [10]. In this paper, we are only interested in double frame, single exposure cameras and all the following discussion is based on this type of digital PIV system.The frame grabber gets the two images from the camera and stores them in the memory so that the processing unit can analyze the movement of small particles in different locations. The results from the processing unit are sent to the output where the data can be used for flow movement analysis or feedback control.

The images recorded by the camera are divided into

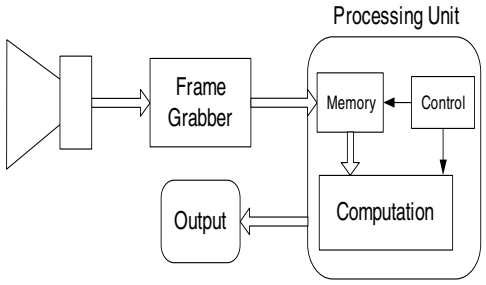

Figure 6. Basic PIV system

small subareas called ”interrogation areas” or ”interrogation windows”. The local displacement vector for the images of the tracer particles of the first and second images is determined for each interrogation area by means of statistical methods. The increasing resolution of CCD sensors makes it possible to get digital images and use cross correlation to find the displacement vector for the interrogation area. The interrogation area can move around and different displacement vectors for the current interrogation area can be computed. Some overlapping of the interrogation area is necessary for accuracy purposes. Typically, the interrogation area from the first and the second images are of different size. In our application, the area taken from the first image is larger than that from the second one. Figure 7 shows the cross correlation process where m, n are the size of the interrogation area from different images and we assume the interrogation areas are square. Correlation moves in raster-scan order: starting in the upper left corner and looping first over columns, and next over rows. Estimation of the particle movement are based on these the cross-correlation results, which is very computation-intensive. Sequential computers cannot meet the requirements of real-time PIV, thus making hardware acceleration necessary for real-time PIV systems.

Digital PIV systems have proved to be useful in several areas. Its applications include aerodynamics [10, 14], liquid flows and even medical research [5]. Nowadays, real-time PIV systems attract more and more interest because of the real-time requirement of some applications. Research as well as application specific commercial systems have been proposed [1, 13]. Tsutomu [9] and Toshihito [4] et al. have proposed a FPGA based real-time PIV system which can process 10 pairs of images in a second hence its performance can satisfy the requirements for real-time processing. However, their approach to interrogation window size is much more restrictive than ours.

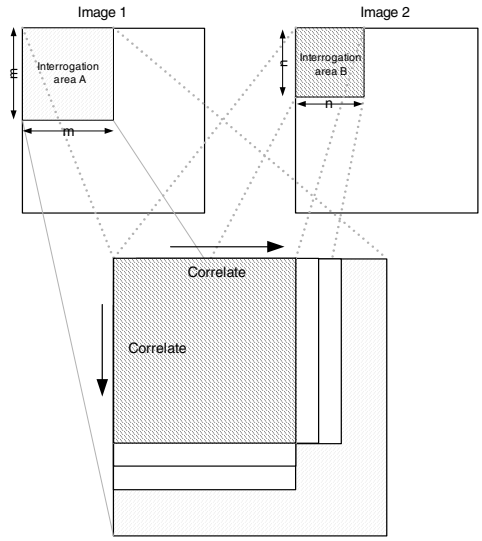

Figure 7. Correlation process of two interrogation area

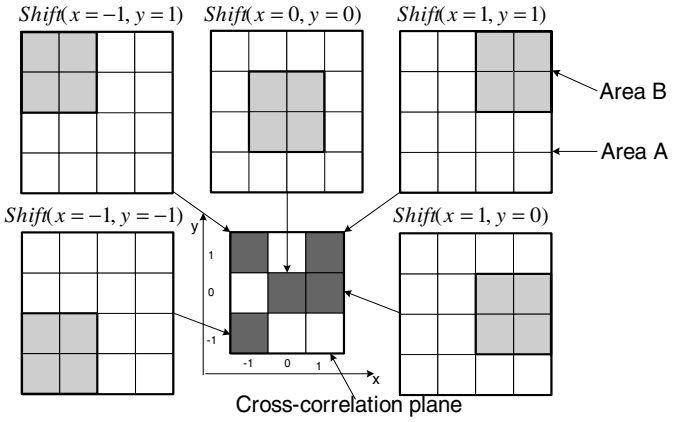

Figure 8. Example of the formation of correlation plane

4.1. Algorithm

4.1.1 Discrete cross-correlation

We assume we are given a pair of same size images containing particle images recorded from a traditional PIV recording camera. The images need to be divided into small interrogation widows. The size of the window is dependent on the application. We call the window from first image *Area A* and that from the second *Area B*. We use N to represent the size of one dimension of the images; m, n to represent the sizes of *Area A* and *Area B*, respectively; and we assume images and interrogation areas are square and $m > n$. Finding the best match between *Area A* and *Area B* can be accomplished through the use of the discrete cross-correlation function, whose integral formulation is given in Equation (1):

$$R_{AB}(x,y) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} A(i+x, j+y)B(i,j) \qquad (1)$$

where *x, y* are called the sample shift. For each choice of sample shift *(x,y)*, the sum of the products of all overlapping pixel intensities produces one cross-correlation value $R_{AB}(x,y)$. By applying this operation for a range of shifts $(-\frac{m-n}{2} \le x \le \frac{m-n}{2}, -\frac{m-n}{2} \le y \le \frac{m-n}{2})$, a correlation plane of the size $(m-n+1) \times (m-n+1)$ is formed. Figure 8 shows an example of correlating two interrogation windows. The size of *Area A* is 4 and *Area B* is 2. We move *Area B* around and match it with *Area A* to find out the displacement of particles recorded in *Area A*. A high cross-correlation value indicates a good match at this sample shift position. The peak value can be used as an estimate of the local particle movement.

4.1.2 Sub-pixel interpolation

After determining the local displacement of particles, we can estimate the movement in pixel shift. Moreover, the position of the correlation peak can be measured to sub-pixel accuracy using sub-pixel interpolation. Several methods of estimating the peak position have been developed. For narrow correlation peaks, using three adjacent values to estimate the correlation peaks is widely used and proven to be efficient. The most common three-point estimators are Parabolic peak fit (Equation (2)) and Gaussian peak fit(Equation(3)). We use Parabolic peak fit in our implementation.

$$\begin{cases} p_x = x + \frac{R_{(x-1,y)} - R_{(x+1,y)}}{2R_{(x-1,y)} - 4R_{(x,y)} + 2R_{(x+1,y)}} \\ p_y = y + \frac{R_{(x,y-1)} - R_{(x,y+1)}}{2R_{(x,y-1)} - 4R_{(x,y)} + 2R_{(x,y+1)}} \end{cases} \qquad (2)$$

$$\begin{cases} p_x = x + \frac{lnR_{(x-1,y)} - lnR_{(x-1,y)}}{2lnR_{(x-1,y)} - 4lnR_{(x,y)} + 2lnR_{(x+1,y)}} \\ p_y = y + \frac{lnR_{(x,y-1)} - lnR_{(x,y+1)}}{2lnR_{(x,y-1)} - 4lnR_{(x,y)} + 2lnR_{(x,y+1)}} \end{cases} \qquad (3)$$

4.2. Hardware System Design

From Equation (1), we know that for one cross-correlation value, we need $n \times n$ multiplication and $n \times n - 1$ addition operations. To determine the peak correlation value for the current interrogation window, we need to calculate all correlation values on the correlation plane. This requires $(m-n+1) \times (m-n+1) \times n \times n$ multiplications

and $(m-n+1)\times(m-n+1)\times(n\times n-1)$ additions. Therefore, if we want to determine the local movement throughout the image, the total computation would be $O(N^2n^2)$ if we assume $m \sim n$. With such a large amount of computation, software cannot process images in real-time in most cases. Hardware implementation is a good candidate for real-time applications since the cross-correlation computation is highly parallelizable. Different applications have different requirements for the size of the image, the size and number of interrogation areas, etc. A specific application may need to change the size of the interrogation area according to the flow movement. ASICs may not be flexible enough to adjust to different design requirements. Thus, FPGAs are a good candidate hardware platform for real-time PIV applications. In the remainder of this section, we discuss our FPGA implementation of the digital PIV system.

4.2.1 PIV System

This design fits into the smart camera system(Figure 1), which provides a memory interface directly from the camera to the FPGA board. As a result, we do not need to move data around unnecessarily before processing. By implementing the PIV processing algorithm into the *Image Processing Part*, we have a PIV system based on reconfigurable hardware. A big advantage of this approach is that processing can start before a complete frame of data has been acquired.

4.2.2 Processing Unit

Cross-correlation is suitable for parallel processing. All the multiplication operations are pixel by pixel and independent of one another and therefore can be computed simultaneously given sufficient hardware resources. The accumulation process can be pipelined by several stages of adders. The hardware structure of cross-correlation is shown in Figure 9. The level of parallelization of the multipliers and the number of stages of adders depends on the hardware resources we can spend on the design. Another important aspect of FPGA design is the memory interface. On-chip memory is fast but has limited size while on-board memory is slow but has large size. As we mentioned in the previous section, at any given time we only need two small interrogation windows to calculate a cross-correlation plane. The memory locations of these two windows can be reused for the whole plane, which means the computation has a *spatial locality* property. In this case we can use a two-stage memory architecture, similar in concept to a *data cache*, to accelerate the processing. Two on-chip memories are used to temporarily store the pixel values of *Area A* and *Area B*. The cross-correlation computation uses these two on-chip memories as input and another on-chip memory as output

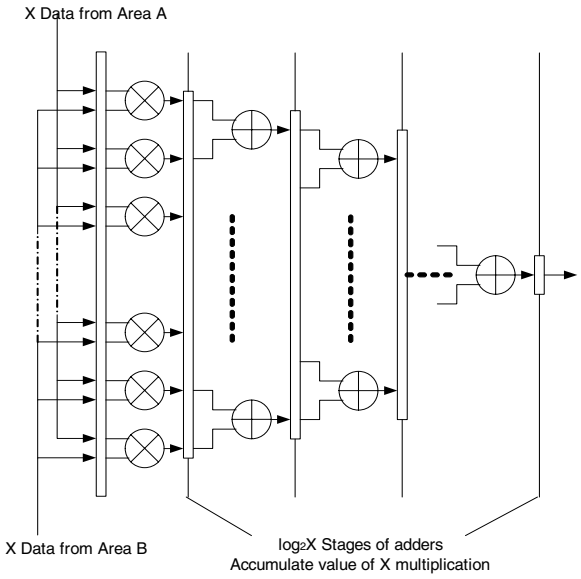

Figure 9. Block diagram of image processing unit

to store the cross-correlation value. The peak value and its position can be recorded for sub-pixel interpolation. Figure 10 shows a more detailed block diagram of the image processing part.

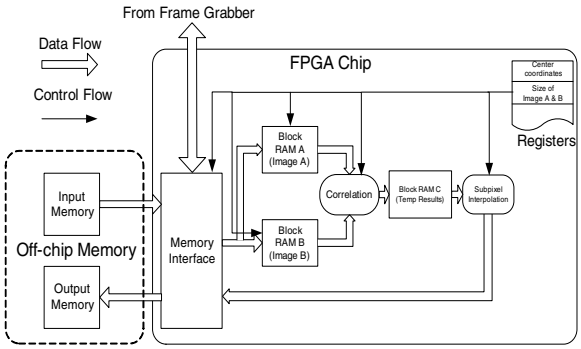

Figure 10. Block diagram of image processing unit

4.3. Results

Our FPGA implementation improves performance in two ways. First, the parallel and pipelined design greatly reduces the image processing time. Second, by using this smart camera system, once we get the data from the frame grabber, we can start processing immediately. Processing can even start before an entire pair of images are captured.

Our application has two input images of size 1008×1016. The interrogation window of Area A is 40×40 and of Area B, 32×32. Interrogation windows have 50% overlap. The sub-pixel interpolation uses the Parabolic peak fit algorithm.

Our results show that for the same cross-correlation and sub-pixel interpolation algorithm, software using fixed-point running on an Intel(R) 1.5GHz Xeon requires 3.4 seconds while the FPGA implementation using an Annapolis Microsystems Firebird board takes only 0.16 seconds. The speedup of data transfer is not so easy to estimate since it depends on the memory type, the way data is transferred etc. But we can safely say that the speedup is more than 20 times for our current hardware structure. This speedup can be further improved with a more parallel structure.

5. Conclusions

We have demonstrated the use of a smart camera setup for two very different applications: medical image processing and computational fluid dynamics. Both applications benefit from processing streaming data on reconfigurable hardware straight from a video camera. In both cases, we have speedups of over 20 times compared to software running on a 1.5 GHz Pentium processor. This setup has several advantages, including minimizing the overhead of data movement in the system, and minimizing processing latency by starting data processing before a full frame of data has been acquired. We have demonstrated using a common memory interface to enable these two different applications.

In the future we plan to apply our smart camera approach to several new applications. We will build up a library of memory interface and component designs that will enable the quick use of a smart camera for real time image processing in a variety of different applications. This will enable new real-time applications of reconfigurable hardware processing at video data rates.

Acknowledgements

This research was supported in part by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation (award number EEC-9986821) and by National Science Foundation Grant CCR-0208791.

References

[1] E. B. Arik and J. Carr. Digital particle image velocimetry system for real-time wind tunnel measurements. *ICIASF '97*, pages 267–277, Sept. 1997.

[2] A. Can, H. Shen, J. N. Turner, H. L. Tanenbaum, and B. Roysam. Rapid automated tracing and feature extraction from retinal fundus images using direct exploratory algorithms. *IEEE Transactions on Information Technology in Biomedicine*, 3(1), March 1999.

[3] E. Cerro-Prada and P. B. James-Roxby. High speed low level image processing on FPGAs using distributed arithmetic. In R. W. Hartenstein and A. Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 436–440. Springer-Verlag, Berlin, / 1998.

[4] T. Fujiwara, K. Fujimoto, and T. Maruyama. A real-time visualization system for piv. *FPL2003*, pages 437–447, Sept. 2003.

[5] P. Hochareon, K. Manning, A. Fontaine, S. Deutsch, and J. Tarbell. Development of high resolution particle image velocimetry for use in artifi cial heart research. *EMBS/BMES Conference*, pages 1591–1592, Oct. 2002.

[6] J. Jean, X. Guo, F. Wang, L. Song, and Y. Zhang. A study of mapping generalized sliding window operations on reconfi gurable computers. In *ERSA '03: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, June 2003.

[7] X. Liang and J. Jean. Mapping of generalized template matching onto reconfi gurable computers. *IEEE Transactions on Very Large Scale Integration Systems*, 11(3), June 2003.

[8] G. Lienhart, R. Manner, R. Lay, and K. H. Noffz. An fpga-based video compressor for h.263 compatible bit streams. In *FPGA '01: ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2001.

[9] T. Maruyama, Y. Yamaguchi, and A. Kawase. An approach to real-time visualization of piv method with fpga. *FPL2001*, pages 601–606, Jan. 2001.

[10] M. Raffel, C. Willert, and J. Kompenehans. *Particle Image Velocimetry*. Springer-Verlag, Berlin, Germany, 1998.

[11] A. Sartori. A smart camera. In W. L. Will Moore, editor, *FPGAs*, chapter 6.6, pages 353–362. Abingdon EE and CS Books, Abingdon, England, 1991.

[12] S. Scalera, M. Falco, and B. Nelson. A reconfi gurable computing architecture for microsensors. In *FCCM '00 Preliminary Proceedings*, Napa, CA, April 2000. Field-Programmable Custom Computing Machines.

[13] S. Siegel, K. Cohen, and T. E. McLaughlin. Real-time particle image velocimetry for closed-loop fbw control studies. *41th AIAA Aerospace Sciences Meeting*, 2003.

[14] C. Willert, M. Raffel, and J. Kompenhans. Recent applications of particle image velocimetry in large-scale industrial wind tunnels. *ICIASF '97*, pages 258–266, Sept. 1997.