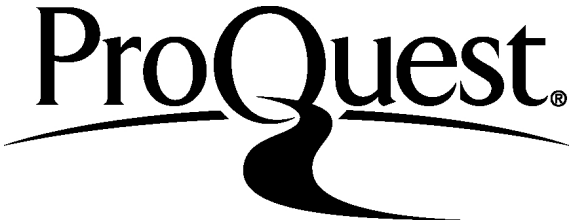
REAL TIME MITIGATION OF ATMOSPHERIC TURBULENCE IN LONG DISTANCE IMAGING USING THE LUCKY REGION FUSION ALGORITHM WITH FPGA AND GPU HARDWARE ACCELERATION

by

Christopher Robert Jackson

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2015



REAL TIME MITIGATION OF ATMOSPHERIC TURBULENCE IN LONG DISTANCE IMAGING USING THE LUCKY REGION FUSION ALGORITHM WITH FPGA AND GPU HARDWARE ACCELERATION

by

Christopher Robert Jackson

Approved: _____
Fouad E. Kiamilev, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

I will start by extending my sincerest thanks to my advisor Fouad Kiamilev. His continued guidance, support, and encouragement inspired me with the confidence to succeed in ways I never would have previously thought possible. I greatly admire his openness, candor, and the friendly atmosphere he promotes in all aspects of life. I am truly grateful for the opportunity he has granted me to join his research group.

I would also like to extend my appreciation to Dean Michael Vaughan, Ms. Veniece Keene, and the Greater Philadelphia Region Louis Stokes Alliance for Minority Participation (LSAMP) for granting me the Bridge to the Doctorate (BTD) Fellowship. I am honored to have been awarded this fellowship. It opened the door for me to pursue this Master's degree, and inspired me to continue toward a Ph.D.

I will also extend additional thanks to Dr. Jiang Liu, Mr. Gary Carhart, and Dr. Mathieu Aubailly of the Intelligent Optics Lab at the U.S. Army Research Laboratory (ARL) Computational and Information Sciences Directorate (CISD) for their partnership and funding of the LRF project through a co-operative agreement from the US ARMY RDECOM under contract number W911NF-11-2-0088. Their patience, direction, and support are an integral part of the continued success of the LRF project.

I want to also thank past and present members of Dr. Kiamilev's research group CVORG at the University of Delaware, who have provided valuable assistance and advice throughout the LRF project. The previous work of David Koeplinger and William Maignan provided the groundwork that made the research presented in this thesis possible. I will also always appreciate contributions and assistance of Brian

Gonzalez, Garrett Ejzak, Kassem Nabha, Tyler Browning, Nick Waite, and Furkan Cayci. Thank you to all of the other members of CVORG for continuously providing a fun, friendly, and interesting environment in which to research.

Last and most importantly, I would like to thank my family. Without their immeasurable love and encouragement, I would never be in a position to pursue my dreams. Words cannot express the depth of my gratitude to them.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

"Lucky-region" fusion (LRF) is a synthetic imaging technique that has proven successful in enhancing the quality of images distorted by atmospheric turbulence. The LRF algorithm selects sharp regions of an image obtained from a series of short exposure frames, and fuses the sharp regions into a final, improved image. In previous research, the LRF algorithm had been implemented on a PC using the C programming language. However, the PC did not have sufficient sequential processing power to handle real-time extraction, processing and reduction required when the LRF algorithm was applied to real-time video from fast, high-resolution image sensors. This thesis describes two hardware implementations of the LRF algorithm to achieve real-time image processing. The first was created with a VIRTEX-7 field programmable gate array (FPGA). The other developed using the graphics processing unit (GPU) of a NVIDIA GeForce GTX 690 video card. The novelty in the FPGA approach is the creation of a "black box" LRF video processing system with a general camera link input, a user controller interface, and a camera link video output. We also describe a custom hardware simulation environment we have built to test the FPGA LRF implementation. The advantage of the GPU approach is significantly improved development time, integration of image stabilization into the system, and comparable atmospheric turbulence mitigation.

Chapter 1

INTRODUCTION

1.1 Background

Often, it is extremely important to be capable of obtaining photographic or video images over long distances. This is particularly vital in military applications including target acquisition, remote surveillance, target tracking, and biometrics. However, long range visual identification and detection is commonly hindered by turbulence due to inclement atmospheric conditions. Atmospheric turbulence causes distortions and warping during imaging. As the distance between the target image and imaging system increases, the effects of atmospheric turbulence increase in severity. Optical turbulence is the result of variations in the refractive index in the path between the camera sensor and the target [1][2][3].

The refraction index of the air varies based on copious atmospheric characteristics including temperature, humidity, and pressure. These fluctuations are seldom homogeneous, such as refractive index changes due to non-uniform temperature distributions. Light waves travelling through such chaotic regions of changing refractive index undergo a complex combination of refraction and scattering, resulting in extensive spatially and temporally varying distortions in the images [4]. Turbulence can be comprised of geometric distortion (motion), spatial and temporal blurs, locally varying blurs, and out-of-focus blurs [5][6]. The atmosphere can also cause differences in the scene from frame to frame, hindering the tracking of moving targets [1].

A variety of image processing methods have been developed in an effort to compensate for atmospheric distortions. Many of these techniques were originally developed for astronomical applications, where it is generally assumed that the distortions are independent of the position in the image plane. This is called the isoplanatic condition [7][8]. An early predecessor to Lucky Region Fusion (LRF) was a method called lucky frame selection, which involved selecting the frames with the highest quality from a set of randomly distorted incoming frames [2][9][10]. However, this approach was not viable under conditions where the isoplanatic condition did not hold, due to a extremely low probability of finding entire frames of high quality[7][10]. Lucky Region Fusion (LRF) was a post-detection technique that was the natural extension of the lucky frame selection approach. Lucky Region Fusion detected portions of incoming frames with high image quality and fused high quality portions from various frames in order to reconstruct the original image even under anisoplanatic conditions[7][11][12][13]. This version of LRF was a post-detection method, like many other similar software based solutions such as speckle imaging or blind-deconvolution [2][3][5]. These methods, when implemented through software on a PC, were generally not capable of the computing power necessary for real-time processing, selection, and reduction of information from real-time visual data [2][11][13]. Military operations usually require observation of the target in real-time.

1.2 Motivation

The LRF algorithm has previously been implemented in software on a workstation PC using the C programming language. This software version of the algorithm was a post-detection implementation, performing the algorithm on pre-recorded images or video over the course of several seconds or minutes. This platform

did not have adequate sequential processing power necessary for real-time image data from high-resolution image sensors [2][7].

The LRF algorithm is straightforward enough to easily lend itself to a parallel implementation. As a result, a natural choice to accelerate the algorithm for real-time video processing was to migrate to a hardware platform. A field programmable gate array (FPGA) is an integrated circuit consisting of hundreds of thousands of configurable logic gates. It is designed to be 'on the fly' configurable by means of a hardware description language (HDL), such as VHDL or Verilog. Moreover, an FPGA has parallel processing capability similar to an application-specific integrated circuit (ASIC). As such, a design implemented on an FPGA has many of the advantages of a software implementation, especially the ability to adapt the algorithm, while also gaining an integrated circuit's parallel processing acceleration. As demonstrated in Figure 1.1, due to parallel processing, the maximum processing bit rate of an FPGA is much higher than that of most sequential processors [2].

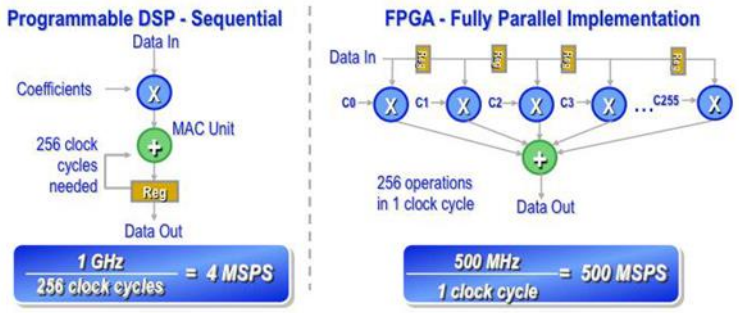


Figure 1.1: Comparison of sequential processing versus parallel processing.

A graphics processing unit (GPU) is similarly capable of parallel processing of large amounts of data for real-time atmospheric turbulence mitigation. Unlike in 2009, when the LRF algorithm was first implemented in software, todays GPUs have the speed, power, memory, and mature programming platforms such as Open Computing Language (OpenCL) necessary to efficiently implement LRF in parallel. Moreover, development using a GPU is generally much faster than attempting to implement the same system using a HDL on an FPGA.

This thesis describes the Lucky Region Fusion (LRF) algorithm, an effective software image processing technique being accelerated by utilizing hardware to achieve real-time image processing. Chapter 2 introduces the LRF algorithm and reviews and details the results of previous work on software and hardware implementations of LRF. It describes the implementation of the algorithm using a field programmable gate array (FPGA) embedded in a high-speed camera, and details the improved implementation of the algorithm where the image processing system is a "black box" with the FPGA separated from the camera. Chapter 3 describes further enhancements to the FPGA "black box" system, including a custom hardware simulation environment, improved user interface and parameter adjustment capabilities, and an approximate solution to the reduced frame rate limitation of the FPGA system. It also goes into extensive detail on the methodology and simulation experiments used during the formation of the final version of the FPGA system. Chapter 4 explores an alternative hardware implementation implemented on a graphics processing unit (GPU) and contains figures displaying and comparing the results of the both the FPGA and GPU LRF implementations.

Chapter 2

PREVIOUS WORK

2.1 Overview of LRF Algorithm

Lucky Region Fusion (LRF) is a multiple frame image restoration technique that has been proven to compensate for atmospheric medium-induced distortions even under anisoplanatic conditions [7]. Unlike other techniques which attempt to mitigate turbulence by modelling or characterizing the turbulence itself, multi-frame image restoration techniques such as LRF strive to estimate the original image by processing a series of degraded images [14]. In particular, LRF is a method which takes advantage of the chaotic spatial and temporal variations in image quality caused by turbulence. Within a set of distorted images, small sections of an image will often be of high resolution for short periods of time. These high-resolution regions are known as "lucky regions". The Lucky Region Fusion algorithm locates, captures, and combines the regions with the best image quality at high speed in order to form a 'fused' image with improved image quality [2][7][15].

The LRF algorithm attempts to improve total image quality through three major steps:

1. Compute the image quality map (IQM) for each incoming image. An IQM is a spatial representation that quantitatively describes the clarity of an image. In the current implementation of the algorithm, it is determined by the sharpness of the edges of objects in the image, along with the distribution of those edges.

2. Compare the IQM of each incoming image to the IQM of current 'fused' image to determine which regions are clearer in the incoming image, if any. If no regions are clearer in the incoming frame, it is discarded.

3. Merge the selected clear regions into the fused video stream. Repeat these steps for each new incoming frame.

2.2 Implementation Procedure for FPGA

2.2.1 Algorithm Mathematics for Hardware LRF

Implementation of the hardware algorithm followed a series of steps equivalent to those presented in Section 2.1. First, an image quality metric was defined. There are many viable choices for an image quality metric. The vector $\boldsymbol{r} = \{x, y\}$ represents spatial coordinates. If we define $I_n(\boldsymbol{r})$ to be the input source stream, then one could define an image quality metric $Q_n(\boldsymbol{r})$ by gradient or by square-intensity [2][7][15]. This image quality metric coincides with the sharpness function introduced in Ref [16] "Real-time correction of atmospherically degraded telescope images through image sharpening." [12]

(1)

$$Q_n(\boldsymbol{r}) = |\nabla I_n(\boldsymbol{r})| \; or \;\; Q_n(\boldsymbol{r}) = \; I_n^2(\boldsymbol{r})$$

The primary advantage of an edge-image detection system is that it permits estimation of the image quality metric without direct calculation of the input image spatial derivatives.12 The Sobel edge detection operator was chosen as the basis image quality metric for the LRF hardware implementation due to its relative ease in implementation using hardware. This is an approximation of a gradient image quality metric

Next, this image quality metric was used to compute the image quality map (IQM). The IQM quantifies the local quality of an image within a region of radius a centered around point **r**. Formally, the IQM is defined as the convolution of the above spatially varying image quality metric, $Q_n(\boldsymbol{r})$, and a kernel $K_a(\boldsymbol{r})$. (typically Gaussian,

$K_a(\boldsymbol{r}) = -exp(x^2 + y^2)/a^2)$, where a is an important scalar quantity known as the kernel size or kernel radius [7][15].

(2)

$$M_n(\boldsymbol{r}) = Q_n(\boldsymbol{r}) * K_a(\boldsymbol{r})$$

Finally, the selected regions with higher IQM are merged into the final fused image [7][15].

(3)

$$I_F(\boldsymbol{r}) = \frac{\sum_n M_n(\boldsymbol{r}) I_n(\boldsymbol{r})}{\sum_n M_n(\boldsymbol{r})}$$

Figure 2.1 shows the above process, in parallel for a frame buffer containing N frames.

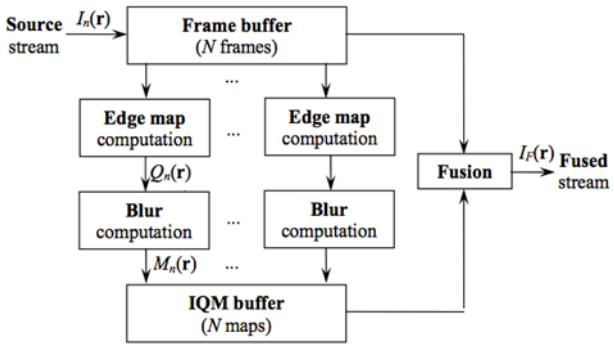


Figure 2.1: Block diagram for hardware LRF algorithm. Edges are found and blurred to produce image quality maps. Higher quality regions are fused into the output stream.

2.2.2 Detailed Hardware Implementation of Algorithm

In section 2.2.1, a general theoretical overview of the steps to implement LRF was presented. Here, the details of each step are expanded upon. Although there have been two generations of the LRF FPGA hardware acceleration platform (see below), both platforms perform the major steps of the algorithm in similar ways.

LRF processing begins by first computing an image quality metric using the Sobel edge detection operator. Edge detection operators such as Sobel are easy to implement using hardware. The Sobel method computes the derivatives of the signal intensity in both the x and y directions to determine the areas where the intensity changes most rapidly [2]. The Sobel operator approximates these derivatives by convolving the image with two 3x3 convolution masks, one for the x direction and another for the y direction [17][18]. Representing these partial derivatives by D_x and D_y, an ideal Sobel edge detector would calculate the magnitude of the derivative of the image pixel as $|D| = \sqrt{(D_x^2 + D_y^2)}$. Implementing a square root operation in hardware can be prohibitively expensive because a square root is solved by iterative approximation [19]. As a result, the initial version of the LRF hardware acceleration algorithm simply used the approximation $|D| = |D_x| + |D_y|$, which is a sufficient approximation in most scenarios [2]. In the second generation LRF system, this approximation was improved by using $|D| = (123/128) * |max(D_x, D_y)| + (51/128) * |min(D_x,.D_y)|$. Note that since the coefficients of the above equation can be represented as fractions of 1024, they are easy to implement in hardware using simple bit shifting. This function has an average error of 2.5% and a maximum error of approximately 5% compared to the true formula [19]. Note that because the pixels on the edge of the image do not have a complete set of neighboring pixels, those pixels needed to be treated separately [20]. This was accomplished by duplicating the lines of pixels near the edges of the image and then applying the Sobel mask normally. The result of the Sobel edge detection on each incoming image is a frame known as the Edge Map. (See Figure 2.2b)

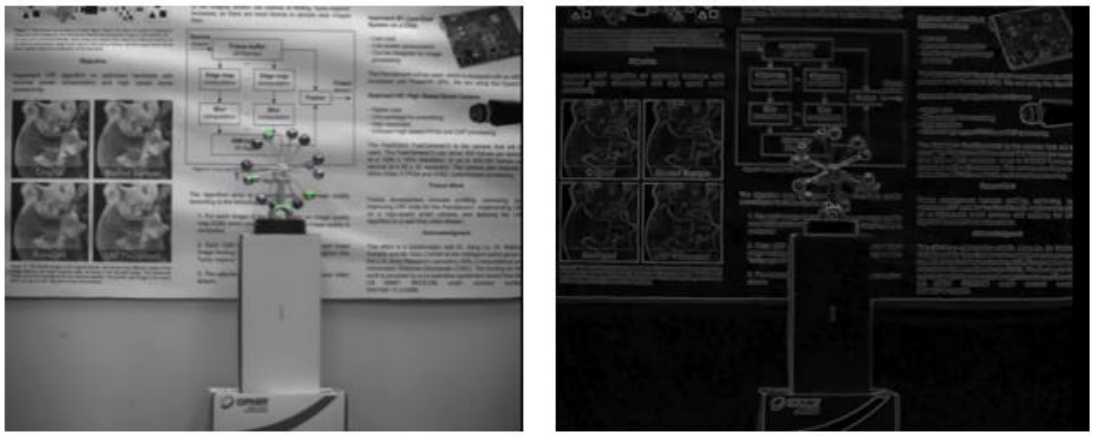


Figure 2.2: Examples of real-time LRF processing with the first generation system. (a) (*left*) **Live** image before processing. (b) (*right*) **Edge Map** created using Sobel Edge Detection.

Figure 2.3: Examples of real-time LRF processing using the first generation system. (a) (*left*) **IQM**. (b) (*right*) **Keep Map**. Bright regions represent high values on the keep map, whereas dark regions are low values. This image primarily consists of bright pixels, so there is very little turbulence present to be corrected.

In order to compute the image quality map (IQM), recall from section 2.2.1 that the image quality metric (in this case the Edge Map) must be convolved with a

kernel of radius a. In the initial version of the hardware system (first generation, see section 3.2), this was accomplished by averaging each pixel in the edge map by a one-dimensional 15-pixel mask (radius $a = 7$), in the horizontal direction and in the vertical direction [2]. This is essentially the equivalent of the two-dimensional mean filter of radius *a*. Ideally, as discussed in Section 2.2.1, we would use a Gaussian filter, but the mean filter was much faster to implement in hardware and consumed less computational resources. The second generation of the FPGA system performs the same calculation, except that the radius is adjustable. This computation produces a blurred version of the edge map, which is used as the IQM. The next step of the algorithm compares the IQM of the incoming ('live') image and the cumulative fused ('synthetic') image in order to determine the lucky regions. The IQMs of both the live and synthetic images are compared one pixel at a time to produce a binary frame. Each entry in this frame corresponds to one pair of pixels, and has the value 1 if the IQM of the live pixel's intensity is greater than or equal to that of the synthetic pixel or 0 otherwise [2]. This binary array is then blurred using a 5x5 convolution mask to produce a frame called the Keep Map. Each value in the Keep Map is between 0 and 25, where a value of 25 represents a region where the incoming live image is of significantly higher resolution than the synthetic image, whereas a value of 0 represents the opposite extreme. Figure 2.4 shows a graphical illustration of this concept. The Keep Map is used to determine the proportion of the new synthetic pixel which will be derived from the new live image and what proportion will come from the previous synthetic image [2] (See Figures 2.3a and 2.3b).

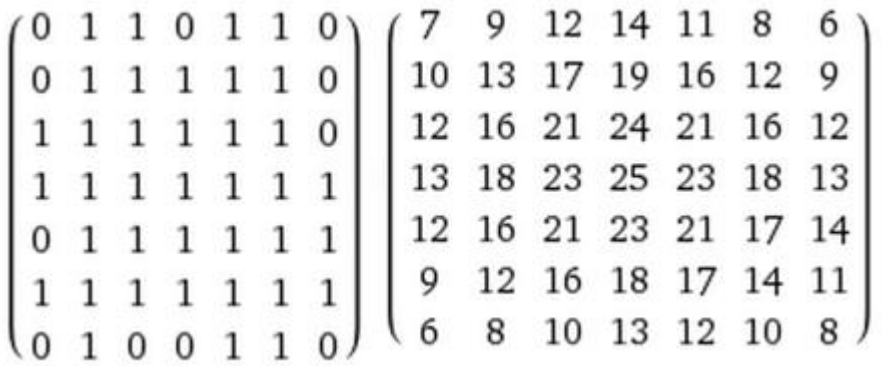


Figure 2.4: Keep Map calculation example. Array on left is binary array where 1 represents a pixel where Live frame is sharper than synthetic frame, and vice-versa for 0. Array on the left is the Keep Map after applying 5x5 blur to the array on the left. Note that values outside the edges are treated as zeros only for the purposes of this example.

2.3 First Generation System

The first generation of the FPGA hardware accelerated LRF system was developed as a proof of concept. A FastVision FastCamera13 was used to capture in the incoming frames at a high frame rate. The FastCamera13 contained a Micron MI-MV13 1.3 megapixel CMOS sensor capable of sending 10 pixels per clock cycle at 66 MHz to deliver up to 500 frames per second at the camera's full 1280x1024 resolution. The FastCamera13 smart camera was also chosen because it contained a Xilinx Virtex II FPGA, which was used to implement the LRF algorithm. The experimental setup utilized a baseboard heater to simulate turbulence as the camera acquired images from a target at a distance of approximately 10 meters.

The first generation of the LRF system had several important limitations. The Virtex II FPGA within the smart camera was released by Xilinx in 2002. As a result, it only contained 2160 Kbits of block RAM (BRAM), which was used by the LRF algorithm for frame storage. To compensate for this limited memory, which could

only store one frame at a time, the first generation system would perform the LRF computation and continuously update the single synthetic frame. When the number of fused frames, N, reached a certain value, the system would output the fused frame to the frame grabber and the process would begin again. Typically N was set to 30, which was empirically determined to be the average value necessary to achieve visible improvement with the experimental setup. Due to this memory limitation, this version of the LRF system was limited to an output frame rate of 1/30 the input frame rate. (See Figure 2.5). Other limitations of this system were its dependency on the FastCamera13, and a fixed IQM blur radius.

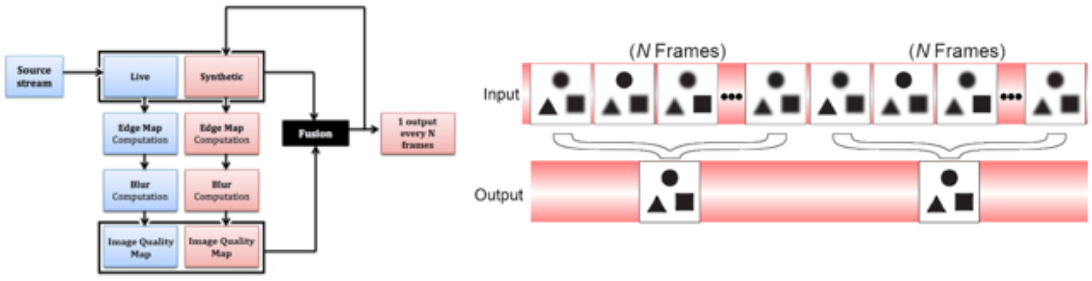


Figure 2.5: (a) Block diagram. (b) Visual representation of 1/N frame rate limitation.

2.4 Second Generation System

The second generation FPGA LRF hardware acceleration system adopted a "black box" approach. In this methodology, the LRF algorithm was implemented on a separate FPGA, and the experimental platform containing this FPGA would interact with a separate, independent camera. This removed all dependency on the FastCamera13 while simultaneously enabling a wider choice of potential cameras or FPGAs. The experimental black box setup was primarily composed of the Xilinx

VC709 connectivity kit with a Virtex 7 FPGA, and a Toyon Boccaccio FPGA FMC Mezzanine card to interface with any camera that utilized a Camera Link interface. The Virtex 7 FPGA on the VC709 has 52,920 Kbits of BRAM. The VC709 also was equipped with two 4 GB 1866 MTs DDR3 SODIMMs. Experiments were run by connecting the LRF system to a Basler aCA 340-km high speed camera, and configuring it to 512x512 resolution, 100 frames per second (maximum 444 fps). Data from the LRF platform was delivered to an Imperx FrameLink Express frame grabber in a PC.

One of the primary limitations of the first generation system was the inability to dynamically adjust the important camera and algorithm parameters, including camera pixel depth, camera exposure time, LRF display mode, and most importantly, IQM kernel size (blur radius). The ability to adjust the fusion kernel size was especially crucial in order to adjust for fluctuations in the imaging medium [3][7][15]. In the second generation system, these parameters were implemented as input signals that could then be adjusted using the VC709 board's pushbuttons and dip switches.

The proposed final version of the second generation LRF system was to implement a circular frame buffer to store N synthetic frames and process the data in parallel. This would allow the system to no longer fuse independent groups of N frames, but instead fuse successive groups of N frames at the full frame rate (see Figure 2.6) [2]. Even with the greater BRAM capacity of the Virtex 7 on the VC709, the BRAM alone is not sufficient to contain the 30 or more frames to be stored in this frame buffer. The LRF system, including a MicroBlaze™ processor and adjustable parameters, uses 30 to 40% of the BRAM resources at any given time while still only storing 1 frame. Consequently, the frame buffer was to be built utilizing the DDR3

SODIMMs on the VC709. Chapter 3 describes the process of building such a system. Diagrams of the planned system can be seen in Figures 6 and 7. Figure 2.7 shows the full block diagram for the LRF system. Figure 2.8 zooms into the LRF processing block, clearly displaying the parallel processing of N frames simultaneously.

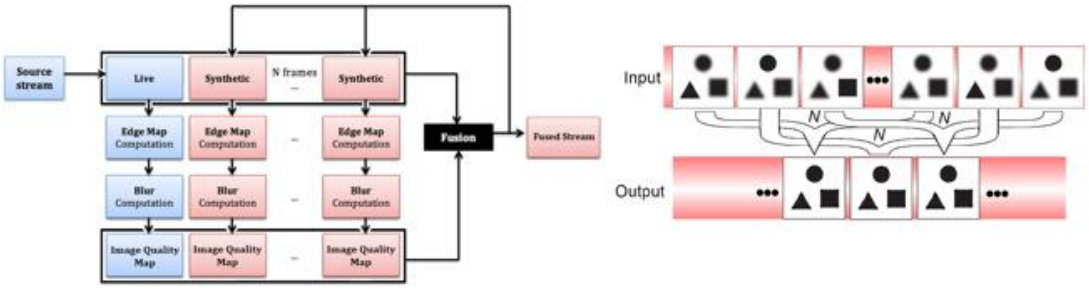


Figure 2.6: (a) Block diagram. (b) Visual representation of circular frame buffer to output frames at the full frame rate.

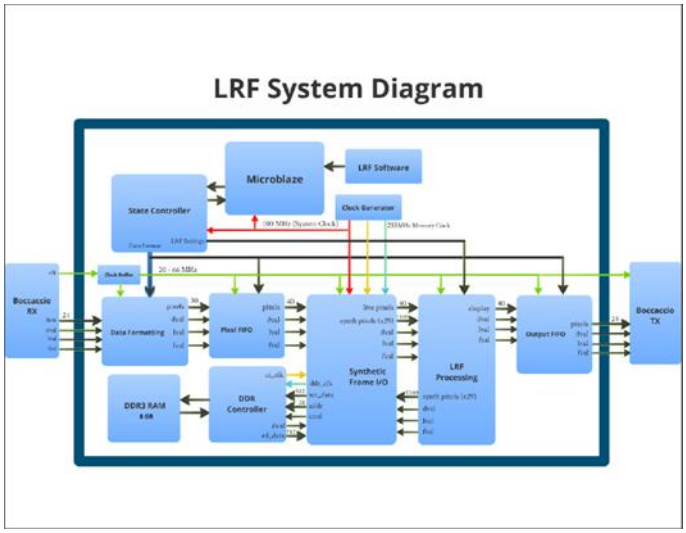


Figure 2.7: System diagram for LRF including a MicroBlaze soft processor and a DDR controller.

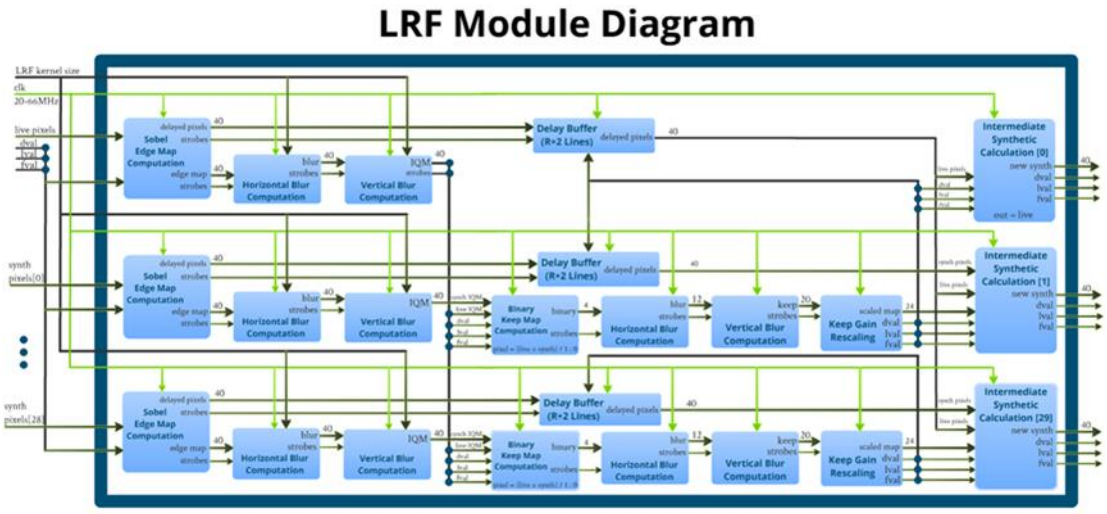


Figure 2.8: Detailed block diagram for second generation LRF algorithm module. This is a close up of the inside of the "LRF Processing" block in Figure 2.7. The operations are performed N times in parallel.

Chapter 3

METHODOLOGY FOR EXTENDING SECOND GENERATION FPGA LRF SYSTEM

3.1 Improvements to the Second Generation System

The proposed second generation system described in Chapter 2 had yet to be fully implemented. The independent "black box" operation using the VC709 FPGA platform was complete. More importantly, this system allowed the user to adjust several crucial LRF algorithm parameters in real-time by means of the board's pushbuttons and dip switches. However, many of the limitations of the first generation system remained. This Chapter details the endeavor of improving the second generation system to reach the planned goal as shown in Section 2.4.

From Section 2.4 of Chapter 2, we described the integration of the ability to alter key algorithm and camera parameters into the second generation LRF system. The VC709 had considerably more resources than the smart camera with the Virtex 2, so it was feasible to create a Xilinx MicroBlaze soft processor to control such parameters. With the MicroBlaze instantiated, it was possible to write a small C program that would run directly on the VC709. These programs were used to write values directly to the registers that corresponded to each LRF parameter that was to be changed. Through serial communication using a PC, a graphical user interface was implemented in both Python and C# to facilitate dynamic adjustment of these parameters. This user interface was also used to serially communicate with the camera to adjust integration time and frame rate as necessary.

Another limitation with the test setup for the previous system was the reliance on a baseboard heater or hotplate to produce artificial 'turbulence' to examine the performance of the LRF hardware algorithm. This was a poor substitute for real

turbulence, which could not easily be produced in a university environment. However, the Intelligence Optics Group at the Army Research Laboratory (ARL) in Adelphi, MD constructed an experimental turbulence setup using a periscope and imaging a water tower at a distance of 2.3 km along a horizontal path about 4° above the horizon. Video data of real turbulence was gathered through this apparatus, and several samples of different turbulence conditions at various times were cataloged throughout the day. An EDT PCI Digital Video Camera Link Simulator was acquired and installed in a PC. This Camera Link Simulator operated like a 'reverse frame grabber', enabling the video of turbulence data to be passed through the VC709 LRF platform as if it were data from a camera. This data was then processed in real-time. Another benefit of this system was the ability to create completely artificial blurred images and pass those through the reverse frame grabber to see how the algorithm performed under theoretical or ideal conditions. An example of the artificial data is seen in Figure 3.1, and an example of the system working on real turbulence can be seen in Figure 3.2 in Section 3.2.

The primary limitation of the first generation system was the reduction of the output frame rate to 1/N of the input frame rate, where N is the number of independent frames fused at a time (typically 30). The first generation output method was to add N frames, then output 1 frame, and then restart the process with new set of N frames. A temporary approximate solution to this reduction in frame rate was devised for the first generation system and implemented on the second generation system. Similar to an IIR filter, the idea was to gracefully decay older frames and continuously output the synthetic frames. This was accomplished by first scaling the Keep Map to vary from a range 0 to 25 to a range of 0 to 128. (The first generation system actually performed a

similar operation, converting to a 0 to 32 range, as operations are much easier for hardware if using powers of 2.) A small adjustable value was added to each value in the Keep Map, but each entry was limited to a maximum of 128. This caused new live data to be very slightly favored over old synthetic data, causing the older data to slowly decay. In this way, the synthetic stream could be output continuously at the full frame rate. Normally, a value between 1 and 4 added to the scaled Keep Map would yield the most promising results. See Figure 3.3 in Section 3.2 for an example using a value of 2.

3.2 Results of Second Generation System before Frame Buffer

In this section, the results from the second generation Lucky Region Fusion system with the improvements described in Section 3.1 are presented. The images in the figures below represent both operation of the algorithm on artificial turbulence and on real turbulence data gathered from the ARL setup described in Section 3.1.

One of the tests for the LRF system was to see how it performed on known, artificial simulation data. The Camera Link Simulator was capable of feeding any stream of images into the LRF FPGA through the Boccaccio Camera Link card. In Figure 3.1, a pattern of random shapes, randomly blurred, was fed into the LRF system. The algorithm did well at clearing up this data.

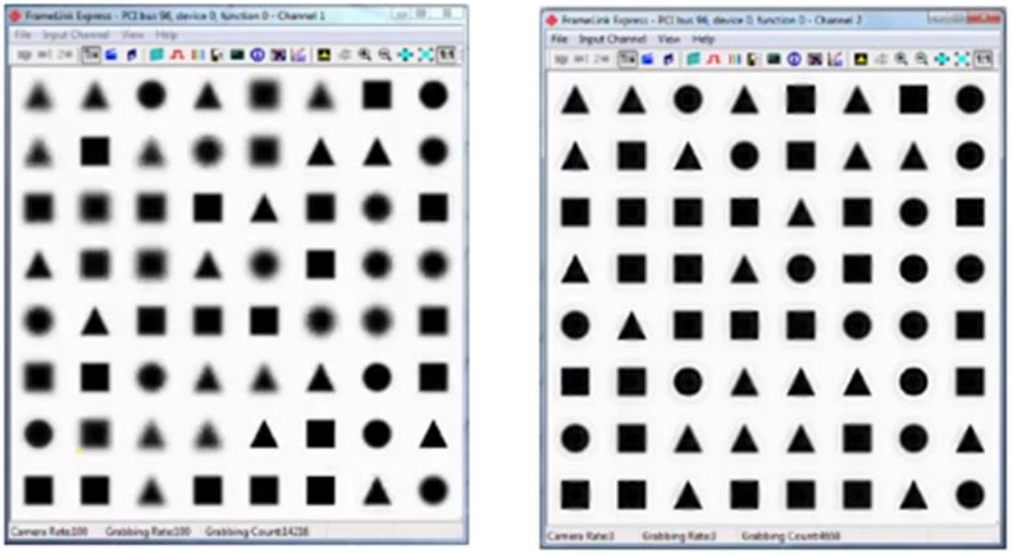


Figure 3.1: Screenshot of algorithm working on a test setup creating synthetic ‘turbulence’. The left video is blurred randomly, with occasional clear regions appearing briefly and randomly. The right video shows the results after LRF processing.

A much more important test of LRF’s capabilities was its performance on real turbulence data. The image data was from a water tower located 2.3 km from the Intelligent Optics Lab at the Army Research Laboratory in Adelphi, MD. At such distances, there was substantial turbulence distorting the image. This data was passed through the Camera Link Simulator into the system. Figure 3.2 shows the results of the LRF processing with the 1/30 frame rate limitation in place.

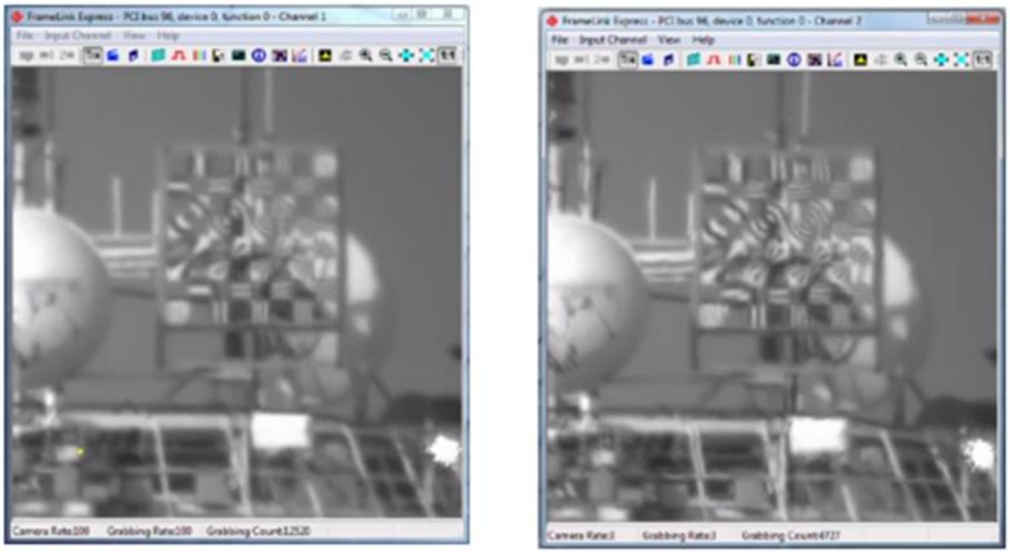


Figure 3.2: Screenshot showing the algorithm at work on a video stream fed through the Camera Link Simulator to the LRF system black box. The left video of a water tower at a distance of 2.3 km imaging through real turbulence. The right video shows the results after LRF processing. Note that the frame rate for the LRF video is only 3 fps compared to 100 fps for the live video, showing the 1/30 frame rate limitation.

As described in Section 3.1, an approximate solution to the frame rate issue was applied to the LRF system. This version decays old synthetic frames gracefully over time, and allows for the continuous output of synthetic frames at the full frame rate. Figure 3.3 shows the results of LRF processing using this method. The output frame rates are the same in both videos.

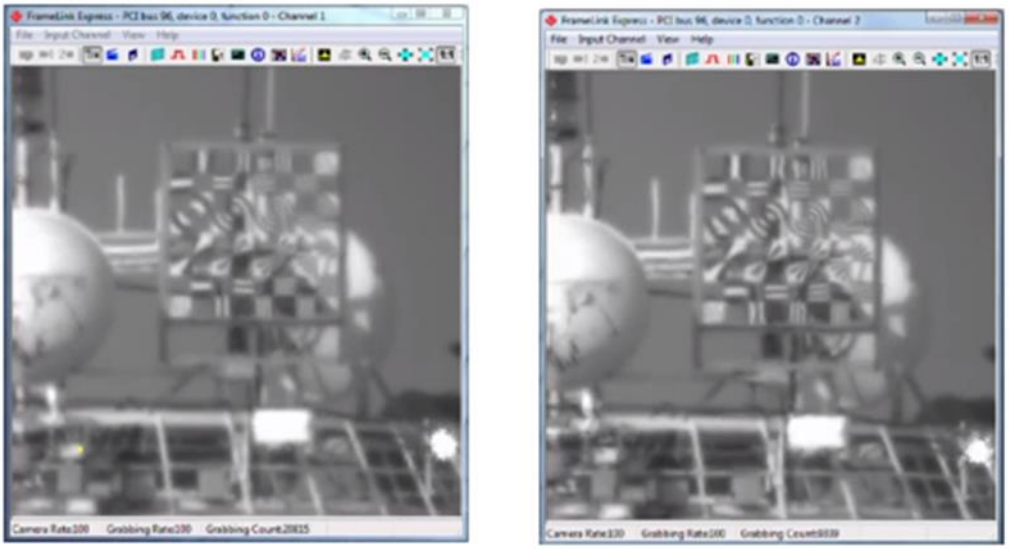


Figure 3.3: Screenshot showing the algorithm at work on a video stream fed through the Camera Link Simulator to the second generation LRF system. The left video is the same water tower at a distance of 2.3 km, imaging through real turbulence. The right video shows the results after LRF processing. The frame rate for the LRF video is now 100 fps, the same as the live video. The 1/30 frame rate limitation has been eliminated.

3.3 Attempts to Integrate DDR to Create Circular Frame Buffer

In order to realize the final version of the second generation FPGA LRF system as described in Section 2.4 and shown in Figure 2.7, a circular frame buffer needed to be built utilizing the VC709's DDR3 SODIMMs. This would provide a more exact solution to the reduced frame rate problem of the first generation system, as opposed to the approximate solution described in Section 3.1. However, interfacing the DDR with the existing LRF Verilog code proved to be far more time-consuming than originally predicted. To facilitate communication with the DDR, Xilinx provided a tool called the Memory Interface Generator (MIG). Additionally, the frame buffer module of the existing LRF Verilog code needed to be altered so that the image pixel data could be sent to the MIG's User Interface (UI) in the expected format.

3.3.1 Characteristics of Xilinx MIG

A Xilinx Virtex 7 FPGA such as that found on the VC709 Connectivity Board includes a memory interface solution core. This core is a combined physical layer and controller to interface the FPGA with DDR3 SDRAM devices. The Xilinx CORE Generator™ tool was used to invoke the MIG to communicate with the DDR3 SODIMMs on the VC709. This extensive set of graphical user interface (GUI) wizard tools was used to properly configure the memory controller for the specific Virtex 7 FPGA and DDR3 SDRAM being used [21].

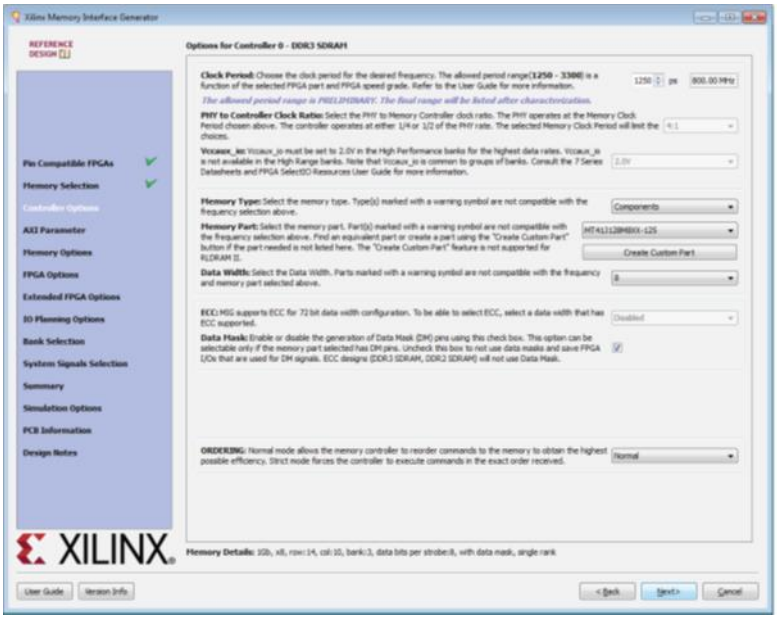


Figure 3.4: Xilinx MIG GUI. [21]

Once the MIG generated the memory interface, it needed to be connected to the controlling application, which in our case was the existing LRF Verilog code. The

memory controller provides a User Interface (UI) that is analogous to a simple FIFO interface, because it always returns the data in order [21].

The MIG UI consisted of three major paths: Command, Write, and Read. Each path would be composed of signals that were asserted by the application (LRF code) and other signals that were asserted by the UI as a result of the current DDR status. For example, the Command path has two signals: the application enable (app_en) signal which is asserted by the user logic (i.e., our Verilog code), and the application ready (app_rdy) signal asserted by the UI [21]. Commands sent when the app_rdy signal was de-asserted would be ignored by the UI, which could result in lost data. Thus, when modifying the LRF frame buffer Verilog module to interface with the MIG UI, it was imperative to produce an app_en signal that remained asserted along with valid application command (app_cmd) and application address (app_addr) values until the app_rdy signal was also asserted.

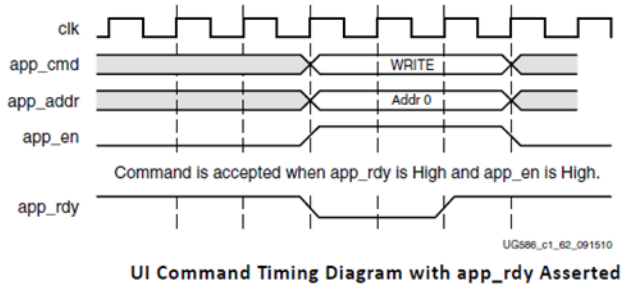


Figure 3.5: Timing for Xilinx MIG UI Command Path. [21]

A similar relationship existed on the Write path between the user logic signals app_wdf_wren and app_wdf_data and the UI signal app_wdf_rdy. Note that app_wdf_data is the actual pixel data that we wish to store into DDR for later use.

After configuring the MIG GUI to use the specific DDR3 SODIMMs found on the VC709 board (Micron MT8KTF51264HZ-1G9E1), the app_wdf_data signal had a maximum data width of 512 bits per clock cycle [21][22]. Furthermore, the MIG was configured as a 4:1 memory controller to DRAM clock ratio with a 64 bit memory at the application interface. This essentially means the memory clock is four times the speed of the FPGA clock. Also, since the memory is DDR (double data rate), it writes on both the positive and negative clock edge of this faster clock. All of this means that behind the scenes, on every FPGA clock cycle when we write 512 bits of app_wdf_data, it is in reality written to the physical DDR in eight 64-bit chunks. This is known as a burst length 8 (BL8) transaction [21].

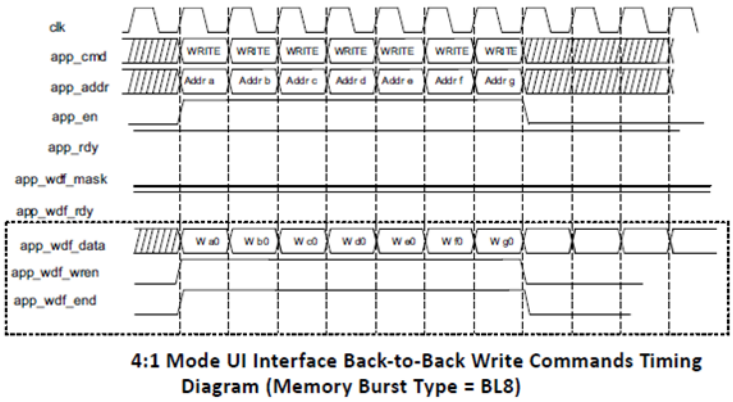


Figure 3.6: Burst Length 8 Timing for Xilinx MIG UI Write Path. [21]

The Read path is the simplest of the three. The read data, app_rd_data, is returned by the UI in order and is only valid when the app_rd_data_valid signal is asserted [21]. The user logic needs only to take into account the app_rd_data_valid signal to make sure it is processing correct data from the DDR SDRAM. Note that like

the app_wdf_data signal, the app_rd_data signal was limited to 512 bits using the VC709's DDR3 SDRAM.

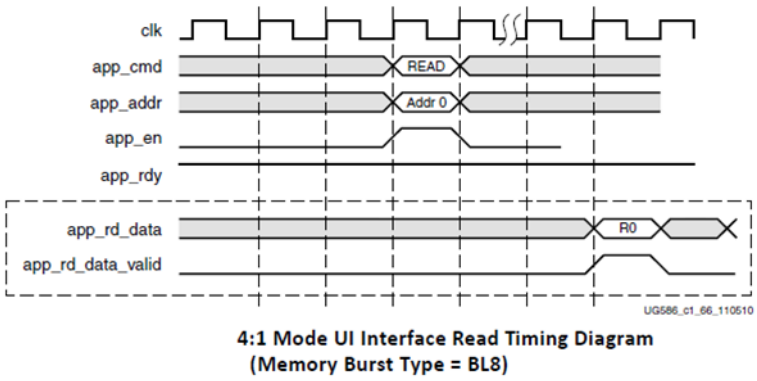


Figure 3.7: Timing for Xilinx MIG UI Read Path. [21]

There are several important aspects of the above discussion of the characteristics of the MIG memory controller. First, the application data (app_wdf_data) must be sent to the UI in sets of 512 bits or smaller. This is actually quite a limiting factor for LRF, because as the preceding chapter stated, we wanted to store 30 or more frames worth of data at a time. Another very important thing to consider about the memory controller UI is that while it is designed to act like a simple FIFO, it is still quite different from a Xilinx BRAM FIFO previously used by LRF. With BRAM, for all intents and purposes the data is available to be read immediately after it is written. However, with DDR, the data becomes available some indeterminate time after it is written. As a result, the UI sends ready and valid signals as discussed above to let the application known when the data is available or when the data being sent back is valid. This in turn meant that it was necessary to rebuild the

LRF frame buffer module to tolerate unknown delays between the writing of data and its availability to be used again, and to utilize the UI ready and valid signals.

3.3.2 Reconstructing the LRF Frame Buffer

The second generation LRF system was described above in Chapter 2. As detailed in that chapter, the LRF code was transferred to the VC709 FPGA platform. Thus, we had a system as seen in Figure 3.8 below. A Camera Link camera sends data through the Boccaccio card into the FPGA in sets of up to 24 bits [23][24]. Those data bits could represent 1-3 eight-bit pixels, or 1-2 ten-bit pixels, or 1-2 twelve-bit pixels, or a single sixteen-bit pixel, depending on how the camera was configured [24]. Although the LRF code could handle any of those configurations, for legacy reasons we typically used two 'taps' of ten-bit pixels, because that was the bit depth of the original FastCamera13 system. We wanted to be able to compare the results of the two generations of LRF easily. In any event, one of the first things done by LRF was to pass the data from the Boccaccio into data formatting and pixel FIFO modules, which converted any pixel data depth from the camera into four sets of ten-bit pixels bits for LRF processing. Similarly, at the output side, there was an output FIFO to convert the 40 bits back down to 24 bits for the Boccaccio, which would then be sent over a Camera Link cable to a compatible frame grabber for display. This version of the LRF algorithm was using the FPGA's BRAM for the frame buffer. Each synthetic frame to be stored could be put into one or more parallel instantiations of BRAM modules, in sets of 40 pixels at a time. However, as stated in Chapter 2, the VC709 had insufficient amounts of BRAM to implement the circular frame buffer necessary to store the 30 or more synthetic frames desired for the final version of the second generation system, as shown in Figure 2-7.

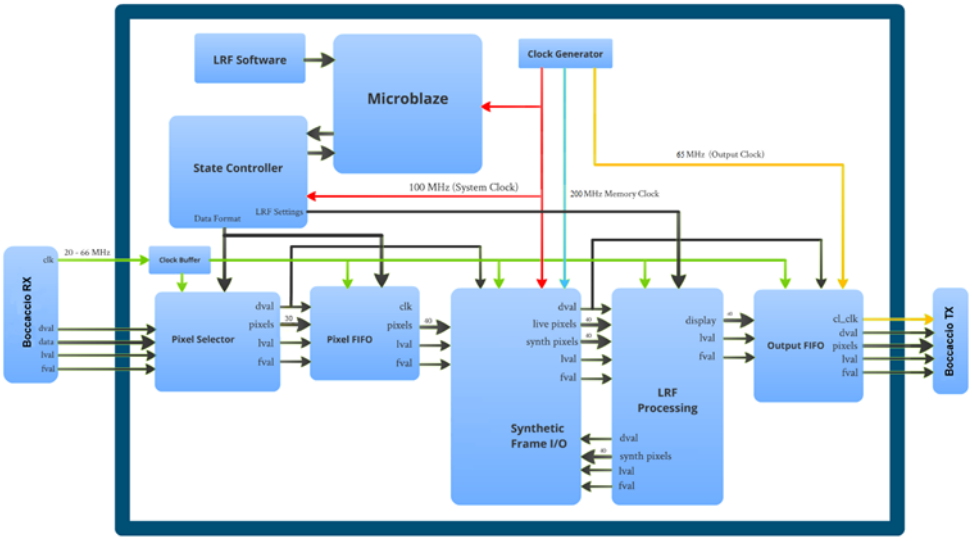


Figure 3.8: LRF Black Box FPGA System before integration of DDR Memory Controller for Circular Frame Buffer.

The strategy for realizing the final version of the second generation system was to replace only the frame buffer module of the LRF code with a new frame buffer module which utilized the VC709's two on-board DDR3 SDRAM blocks. Ideally, the rest of the LRF modules could then remain untouched and continue to operate as they always had. The first attempt at writing a DDR frame buffer module was made by former CVORG researcher David Koeplinger. His original design can be seen below in Figure 3.9.

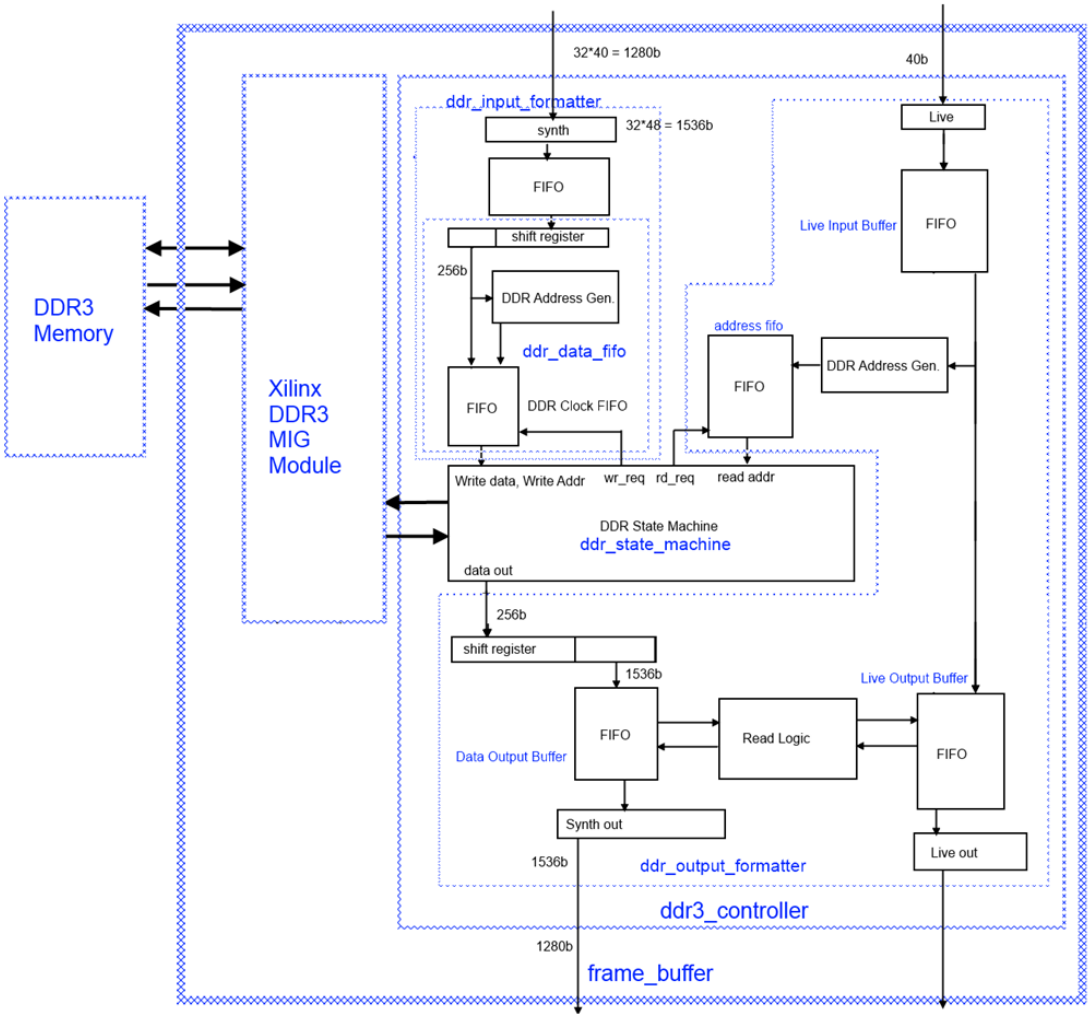


Figure 3.9: Original DDR Frame Buffer Design.

This design was far more complex than the original BRAM frame buffer. A large part of this complexity stems from the discussion in the previous section. Several additional signals needed to be added to the frame buffer module to communicate with the Xilinx DDR3 MIG memory controller module. These include the ready signals (app_rdy, app_wdf_rdy) and valid signals (app_rd_valid), to name a few. Moreover, these signals then needed to be controlled by a state machine, which could control the

flow of data through the system so that, for example, data was not sent to the DDR when the DDR ready signal was de-asserted.

In addition to the MIG and DDR state machine, the frame buffer also included two major blocks: the input formatter and output formatter. The input formatter, output formatter, and state machine together formed a module called the DDR3 controller. Recall from the previous section that one of the major limitations of the input data into the DDR3 MIG memory module was that the width of the app_wdf_data was limited to 512 bits. There were two problems with the 512 bit width size. First, recall that the rest of the LRF system was set up to handle four sets of ten-bit pixels, or 40 bits at a time, which does not factor nicely with 512. Second, and far more important, was the fact that we wanted to handle 30 or more frames of data in parallel for the LRF frame buffer, and 30 * 40 = 1200, which is far greater than 512. To solve these issues, the input and output formatter modules were created. Setting the number of synthetic frames to process simultaneously to 32 yielded a synthetic data width of 40 * 32 = 1280 bits. Then, the input formatter zero-padded each of the ten-bit pixels with two zeros, creating twelve-bit "pixels" for DDR storage, such that 12* 4 = 48 and 48 * 32 = 1536-bit wide synthetic pixel data width. Since 1536 is a multiple of 512, we could then split the synthetic data into three 512-bit parts and send each to the DDR MIG memory module on the app_wdf_data signal. The data was placed into a FIFO so that it could be pulled out by the state machine and sent to the DDR when the DDR was ready to receive it. Similarly, the output formatter read the 512-bit chunks of data from DDR on the app_rd_data signal, stored every three into a signal 1536 bits wide, then truncated off the zero padding to yield a signal 1280 bits wide. This represented 32 * 40 bits, or 32 separate synthetic frames as originally

stored. The output formatter also included logic to synchronize the synthetic frames with the Live frames coming in from the camera, so that each synthetic frame was being compared with the correct Live frame for LRF processing.

3.3.3 Frame Buffer Simulations

Each of the portions of the frame buffer described above was developed and coded in Verilog. There were many different components and signals involved in the frame buffer, so we decided it would be best to run simulations to make sure each component was operating correctly before plugging the whole frame buffer block into LRF. The Xilinx Vivado Design Suite® (version 2013.2) was used both to write the Verilog code for the frame buffer components and to simulate the output using the native Vivado Simulator. We created a test bench that would send simulated "pixels" of data it sets of 32 frames of four ten-bit pixels for the synthetic data, and frame of four ten-bit pixels for the live data, exactly as we would expect from LRF and described in the section above. In other words, the synthetic data width was 1280 bits per pixel clock and the live data width was 40 bits per pixel clock as the inputs to the frame buffer module. It should be noted that the pixel clock is based on the Camera Link standard and the limitations of the Boccaccio FMC card and thus limited to a maximum of 66 MHz, but this is much slower than the FPGA clock of 200 MHz [22][23]. Consequently, the logic manipulating the data for the frame buffer could more than three times faster than the incoming data. This was important, because from the discussion above, the incoming synthetic data needed to be split into three smaller pieces of 512 bits each. The FPGA clock was fast enough to operate on three sets of 512 bits before the next set of data would come in from the simulated test bench "camera".

Simulating the frame buffer module proved to be an arduous and time-consuming process. The Vivado Simulator produces waveform timing diagrams for the signals one wishes to examine, similar to Figure 3.10 below. As can be seen in that Figure, early attempts to produce output data (save_data_out signal) were rather unsuccessful. The first tests involved a complete, but simplified, version of the frame buffer shown in Figure 3.2.The only differences with the simplified version were: (1) the synthetic width was 640 bits rather than 1280 bits and zero-padded to 768 bits rather than 1536 bits, and (2) for early simulation purposes, the DDR was replaced with a BRAM. These simplifications were made to make it easier to detect errors in the output waveforms. The main aspects being testing in these early stages was the splitting of the data into three parts (256 bits each in this case), storing them into BRAM, and then reading them back in the correct order and in sync with the live data.

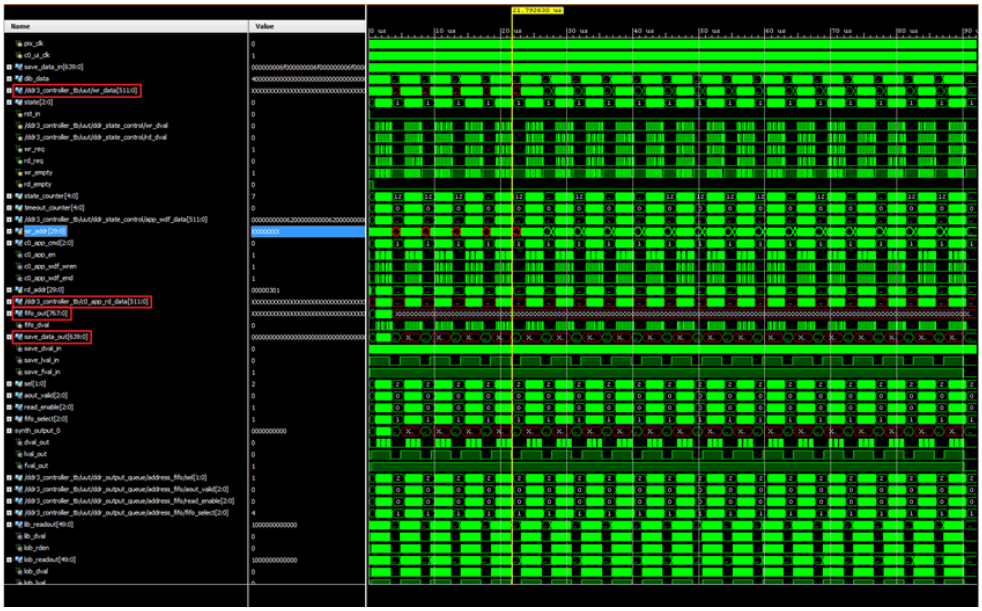


Figure 3.10: Early simulation of frame buffer design.

Testing the early system was time-consuming, because each of the Vivado simulations would require 10 to 30 minutes to complete for waveform windows between 1 and 10 microseconds. Also, it was apparent that the entire frame buffer module was not operating properly, for we were not seeing the correct data at the output, as seen in Figure 3.10 above. It quickly became clear that in order to properly test the system, each of the sub-blocks needed to be tested individually. This process slowly allowed us to find and correct the initial problems. We obtained output data that seemed to be correct as seen on the 512 bit app_rd_data signal in Figure 3.11

Figure 3.11: Correct output data app_rd_data using early frame buffer design.

The next step was to simulate the frame buffer with the full synthetic data width of 1280 bits (representing 32 parallel frames) with a model for the DDR3. When

configured, the Xilinx MIG also generated a simulation model for the DDR3 hardware specified during memory controller generation. This allowed for the monitoring of the important memory controller UI signals, such as app_wdf_data, app_rdy, and app_rd_data. In addition, some of the DDR3 status signals could also be monitored, most notably init_calib_complete, which indicated when the initial calibration of the DDR3 SODIMM hardware was complete. Unfortunately, there were a number of problems with the DDR simulation model that made development extremely slow. We discovered that until the init_calib_complete signal asserts, nothing could be written to or read from the DDR. After some trial and error, it was found that this init_calib_complete signal, even with the simulation's SIM_CAL_OPTION set to "FAST", did not assert until after 47 microseconds on the simulation waveform window (see Figure 3.12). Previously, we were running the BRAM simulations for about 1 to 10 microseconds, which required 10 to 30 minutes of real time to complete. The DDR simulations, consequently, often required between 90 minutes to 2 hours before we could see or test any output at all. As a side note, if the SIM_CAL_OPTION setting was not set to "FAST", the simulation would also simulate the power on startup time and thus require over 500 to 700 microseconds on the waveform for init_calib_complete to assert; this took over 18 hours of real time to simulate! In any event, the typical 90 minute simulation times drastically slowed the testing of the DDR3 version of the frame buffer module.

Figure 3.12: 65 µs simulation. The init_calib_complete signal asserts at 47 µs.

Extensive testing and simulation was done on the frame buffer system with DDR3. Even after repeatedly tweaking and rewriting the memory controller state machine to match the guidelines as described in Section 3.3.1, we often found that the output data would be slightly incorrect. For example, the output data might have a pixel missing, or a pixel might have been repeated on the output unexpectedly, or the data might come out in the incorrect order. In order to narrow down where the potential problems might occur, we decided to modify David's design from Figure 3.9. The modified design can be seen in Figure 3.13. In the new design, we drop the live and synthetic synchronization logic. Instead, we remove one frame from the synthetic side and attach the live data to the end of the input data signal. Thus, instead of a separate 1280 bit synthetic signal representing 32 frames and another single frame of

live data, we have a single 1280-bit input stream representing 31 synthetic frames and 1 live frame. From there, this 1280 bit signal is zero-padded to 1536 bits just as before. This 1536-bit signal is split into 3 parts of 512 bits, each of which enters a dual clock FIFO, with the input clock being the pixel clock (65 MHz) and the output clock being the FPGA UI clock (200 MHz). The state machine would then read each of these FIFOs in turn, based on the ready signals from the MIG controller UI, and write the data to the DDR. The entire process would then be reversed on the read-side, with the data being reconstructed back into a 1536-bit wide signal, then the zero-padding trimmed off to recover the original 1280-bit signal representing the 31 synthetic frames and 1 live frame. This method does lose a single synthetic frame of accuracy on the LRF processing. However, the advantage of automatically syncing the live and synthetic data at the output without any extra logic is significant.

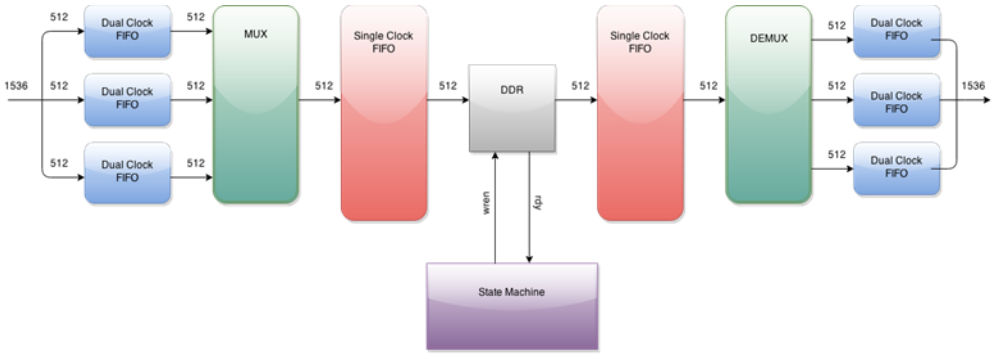


Figure 3.13: Modified frame buffer design and data path.

The modification of the frame buffer module described above eventually reached a point where it appeared to pass all of the simulations. Figure 3.7 shows the correct waveforms from a 65 microsecond simulation. At every point during the

simulation, the output data from the simulated DDR matched the input data. We were then ready to actually replace the frame buffer in the LRF module with this new one to see if similar results could be observed. Fortunately, the LRF code and frame buffer were designed to be modular and it was relatively straightforward to replace the existing buffer with the new one with only a few parameters being changed.

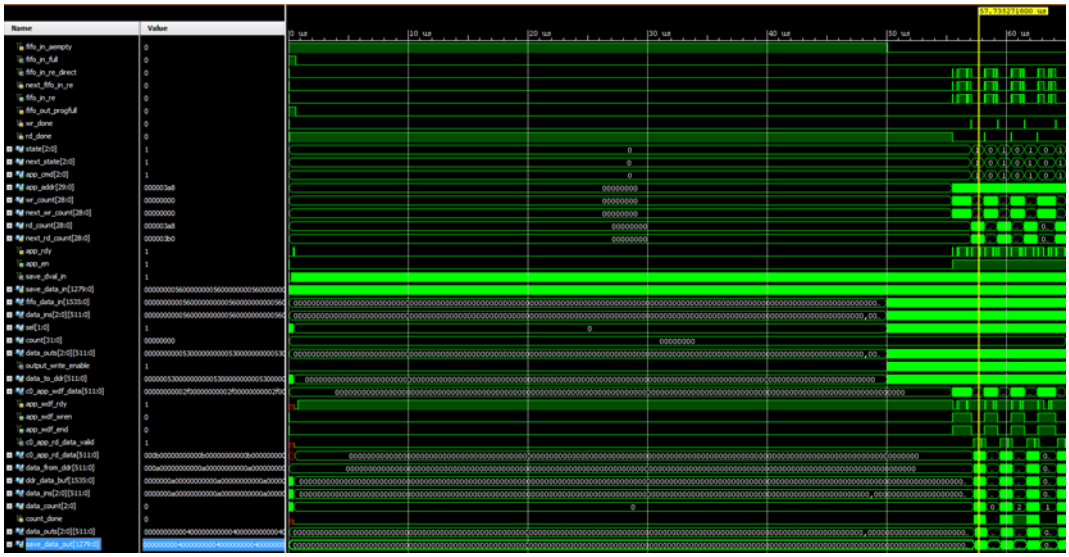


Figure 3.14: 65 μs simulation. The output matches the input.

3.3.4 Integrating Frame Buffer into Existing LRF Code

As with any FPGA design, the LRF system with the new frame buffer needed to go through both the Synthesis and Implementation steps in order to generate a bit file that could be programmed onto the VC709. Early attempts at synthesis required 30 to 60 minutes, and uncovered some unforeseen problems with the combined design. As can be seen in Figure 3.15, the estimated Utilization of the VC709's resources far

exceeded its capabilities. The report shows and estimated 172% of the lookup tables (LUTs) and 218% of the BRAM being required to implement the design.

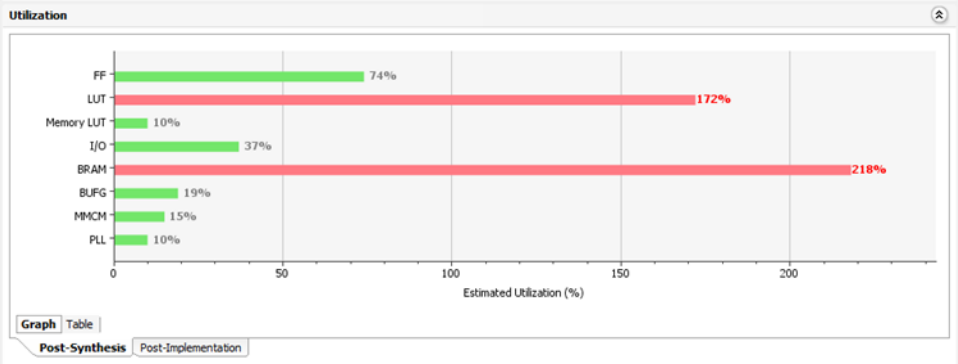


Figure 3.15: Post-synthesis utilization report. Insufficient LUT and BRAM.

What is happening here is the LRF processing of 31 frames is actually far more resource intensive than anticipated. Recall that previously, when the LRF was using BRAM to store a single synthetic frame, over 30% of the BRAM was being utilized. However, from Figure 3.16, notice that the DDR module storing all 30 frames only utilizes 14% of the total BRAM. Therefore, as far as storage is concerned, our design goal was met. That being said, Figure 3.16 also illustrates the problem. Previously, when operating on a single synthetic frame, only about 4 to 6% of the BRAM was being used for LRF calculations such as horizontal and vertical blurring. On the other hand, with the new system, performing these same LRF operations on 31 frames in unison each requires about 4-6% of the total BRAM, which is cumulatively more than the VC709 has available [25].

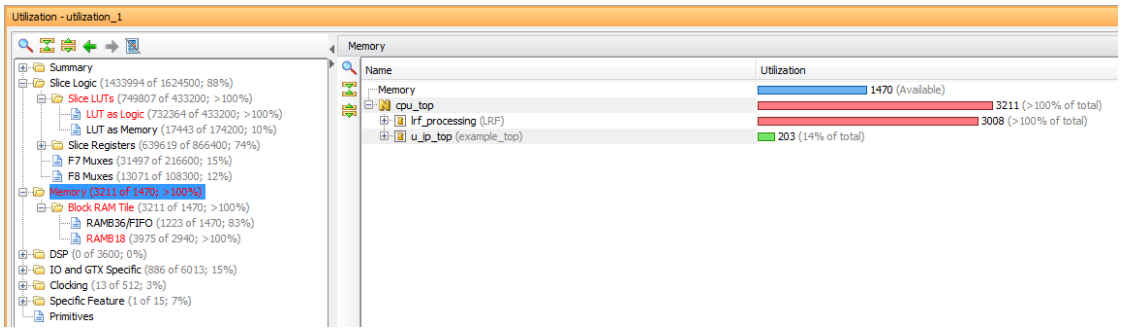


Figure 3.16: Utilization report. DDR module u_ip_top only uses 14% of total

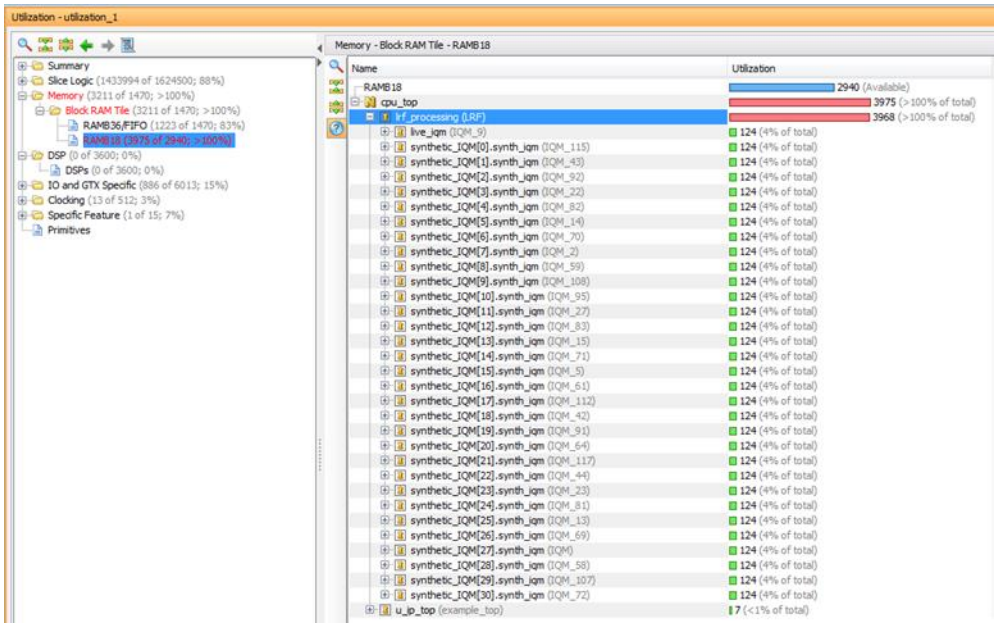


Figure 3.17: Utilization report. Each synthetic IQM calculation requires 4% BRAM.

This was a troubling result, because it meant that even if the DDR storage part of the frame buffer worked perfectly, we would not be able to process 30 or more frames of data simultaneously with a LRF blur radius up to 30, as we could with the single frame system. Nevertheless, we still wanted to know how well the DDR frame buffer could integrate into the system. The total amount of BRAM required for the

synthetic calculations depended on both the LRF radius and the number of synthetic frames. Specifically, the number of 36 Kb block RAM elements required for the expensive LRF vertical blur calculation is given by equation (4). The number of block RAM elements required for the remaining delay buffer calculations is given by equation (5). In these equations, R is the LRF blur radius, and F is the total number of frames to be processed (both synthetic and live).

(4)

$$(2R + 2) \cdot F$$

(5)

$$(R + 2) \cdot F$$

By maximizing equations (4) and (5) while taking into account the total number of BRAM on the VC709's Virtex 7 FPGA (XC7VX690T) being 1470, and the DDR frame buffer consistently requiring 203 (14%) of the BRAM tiles, we found that we could create a system with a radius of 25 and 16 frames (15 synthetic, 1 live) while staying under the BRAM limit [25]. Figures 3.18 confirms that the BRAM usage has been reduced to 99% (and LUTs down to 88%).

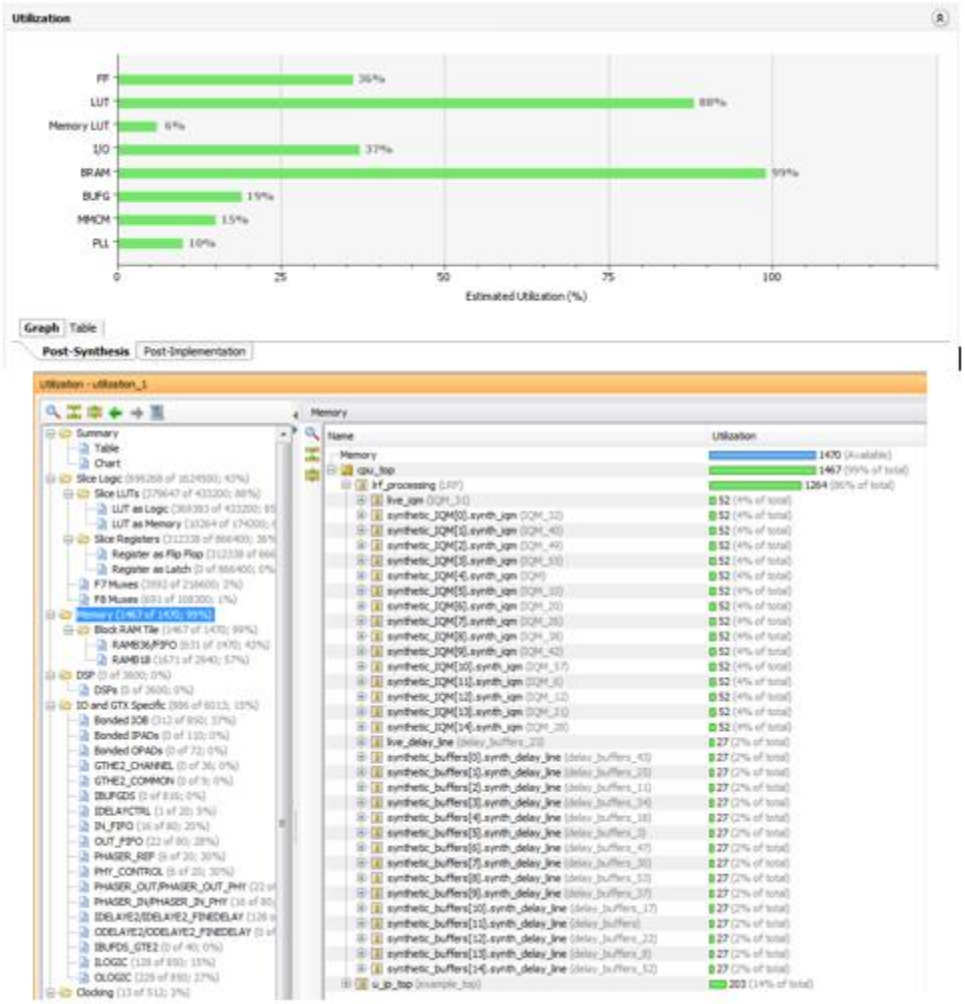


Figure 3.18: Utilization problems corrected.

Finally, with Synthesis complete, we were able to run the Implementation step, which placed and routed the design and generated a bit file to program onto the VC709's FPGA. Implementation required well over 3 hours to complete. Unfortunately, the results on the output for the full system failed to meet the promising results of the simulation. Figure 3.19 shows the output from the camera after LRF processing with the new frame buffer. There is no discernable picture, which is

obviously not good. The most positive aspect of the result is that the output frame rate is correct, that is, we are seeing N frames out for every N frames in. Also, the white streaks in the image does seem to correspond to data, for it changes when objects pass in front of the camera lens, and they vanish when the lens is covered completely.

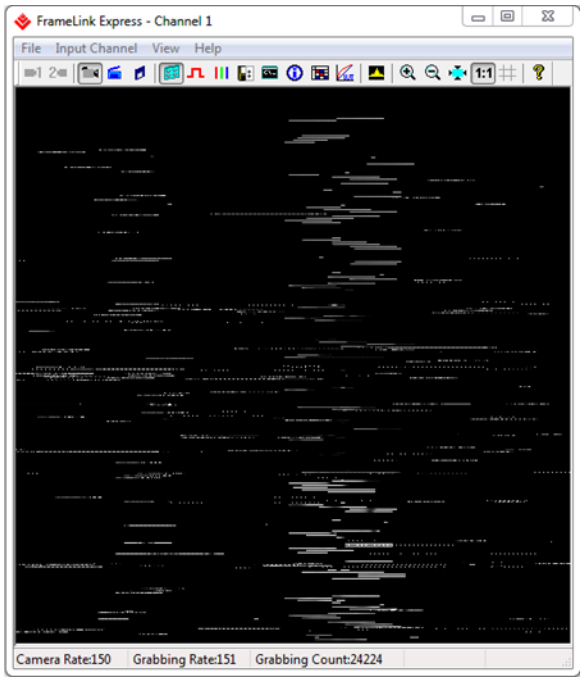


Figure 3.19: Video output after unsuccessful Frame Buffer integration.

We have a number of theories and ideas on why there are problems with the output and what could be done to fix them. For example, the frame buffer input FIFOs might be overflowing at certain points in time causing data to be lost. Another idea involves the valid signals from the Camera Link somehow going out of sync, causing the frame grabber to reconstruct the data incorrectly. However, at this point we realized that the development time on the FPGA was becoming a major hurdle. With

simulations requiring two or more hours, and synthesis/implementation requiring another 3 hours every time we wanted to change anything, a search for another solution seemed prudent. Also, recall that even if the frame buffer module worked perfectly, the VC709 could only support up to a 16 frames at 25 radius for LRF processing. In order to process more frames, we would need to either completely re-optimize the LRF code, or acquire a more powerful FPGA platform. Therefore we decided to explore the option of implementing LRF on a GPU platform instead to see if we could achieve comparable results. That exploration is discussed in Chapter 4.

Chapter 4

LRF GPU IMPLEMENTATION

4.1 Benefits of a GPU Approach

As discussed in Chapter 2, the LRF algorithm has previously been implemented on a workstation PC using the C programming language. However, due to the large amounts of image data processed by LRF, the sequential processing of workstation's CPU platform was insufficient for real-time atmospheric turbulence mitigation. Therefore, parallelization of the algorithm on an FPGA platform was investigated and realized, as discussed in Chapters 2 and 3. At the end of Chapter 3, it became apparent that the simulation and development times involved in expanding the LRF algorithm could quickly become impractical.

There is another method of hardware based processing parallelization common in image processing, the graphics processing unit (GPU). In fact, in many ways a GPU is ideal for the type of processing done by LRF, as GPUs are typically designed to process large numbers of pixels simultaneously. In 2009, when the LRF algorithm was first implemented in software on a workstation PC, a GPU solution was suggested but originally rejected. At the time, parallel programming platforms like Compute Unified Device Architecture (CUDA) were still relatively new. Even when accelerated with a GPU, at the time there was insufficient processing power necessary for real-time image data from fast, high-resolution image sensors.[2,7] In the years since LRF was first implemented in software and the on an FPGA, GPU platforms and programming frameworks have advanced considerably. In principle, the GPU has become a competitive and viable alternative for real-time acceleration of the LRF algorithm.

A significant advantage to working with a GPU is the ability to use high-level frameworks designed for parallel programming of heterogeneous frameworks [26]. Unlike a hardware prototyping language like VHDL or Verilog, the GPU programming frameworks are much closer in structure and syntax to a high-level CPU programming language such as C. This typically means the code is much easier and faster to understand and debug. More importantly, compile times are on par with typical CPU programming languages, which vastly improves development time. Perhaps the most well-known of these frameworks are CUDA and Open Computing Language (OpenCL). CUDA is a couple of years older than OpenCL and generally has more documentation, but created by NVIDIA exclusively for NVIDIA GPUs. Although we are currently using NVIDIA GPUs for testing purposes, we decided to use OpenCL because it is compatible with any GPU.

The OpenCL framework provides a number of useful abstractions to make working with GPUs more efficient. At the most basic level, there are two major components to an OpenCL system, the host, and one or more devices [27]. The host is typically a CPU which runs a high-level programming language such as C (extended by OpenCL). The host schedules and sends data to the device(s) in order for it to be processed in parallel. The host assigns parallel computational tasks, known as kernels, to multiple processing elements in the devices for simultaneous computation [27]. (These GPU kernels are not to be confused with the convolution kernels described in Chapter 2.) A device is a general term for a component capable of parallel processing, and typically refers to a GPU [26][27]. OpenCL further abstracts the device down to portions called compute units, work-groups, and work-items. Work-items are the smallest of these abstractions. Work-items represent an individual execution of a

kernel on a specific set of data. In image processing like LRF, a work-item could represent an operation on an item as small as a single pixel. If the kernel is a task such that each pixel could be calculated individually, such as multiplying the pixel value by some number, then each pixel could be a work-item and all of them could be calculated at once. A work-group is a combination of work-items which all have access to the same computing resources, and a processing resource that can support a work-group is known as a compute unit [27]. All of the work-items in a work-group can be synchronized on the device. Moreover, each work-item in a work group can access a very fast block of memory on the compute unit called local memory [27]. In image processing, a work-group might represent a row or column of pixels, or a small sub-image, or even an entire frame. This work-group and work-item model is extremely efficient for image processing tasks due to this potential for very high data throughputs.

A host sends data and kernels to a device for parallel processing. The data is stored on the device in global memory, which can be accessed by all of the compute units on the device [27]. In optimized OpenCL implementations, portions of the global memory are copied to the much faster local memory, such that the work-items in each work-group only work on the data that they need. Each work-item also has tiny but extremely fast private memory that it uses for intermediate calculations [27]. When the compute items finish the kernel tasks, they send the processed data back to global memory, so that it can then be read back by the host. Figure 4.1 shows the relationship between work-groups and kernels. Figure 4.2 displays the OpenCL device model.

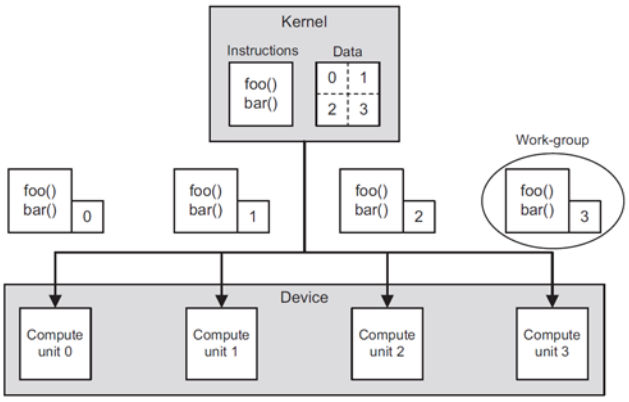


Figure 4.1: OpenCL kernel and work-groups. [27]

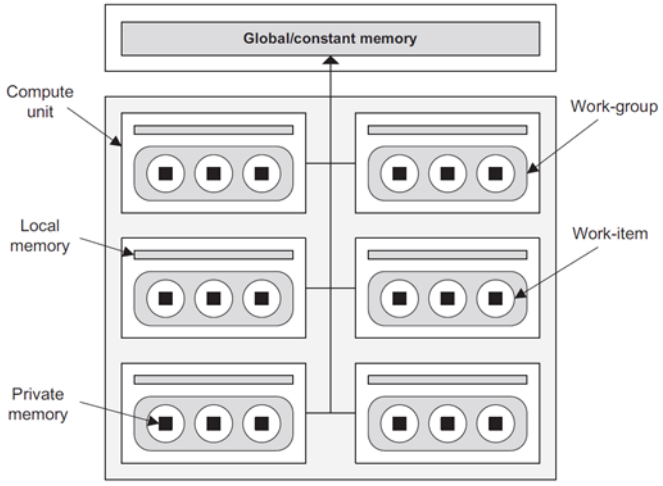


Figure 4.2: OpenCL device and memory model. [27]

4.2 Development of the GPU LRF System

The previous section gave a concise overview of the OpenCL host/device model. For development of the host side code, OpenCL is typically an extension of the C programming language. However, there other options depending on the needs of the developer. Since for LRF our primary concern was faster development times, we opted to use the Python programming language with the PyOpenCL library. In

practice, PyOpenCL is simply a Python wrapper for the underlying OpenCL C code implementation. Run times on Python are usually slower than C since Python being an interpreted language and not a compiled language, but we thought the increased development speed using Python was worth the tradeoff. We were attempting to determine if a GPU LRF implementation was viable or even possible. However, as noted in Chapter 5, future work involves complete optimization of the GPU code, including porting it to C.

Initial development of the GPU version of the LRF algorithm was done using recorded image data of the water tower as shown in Section 3.2. Recall that various sets of this turbulent image data were recorded over the course of the day, allowing for the testing of the algorithm in different turbulence conditions. Use of the Python version of the Open Computer Vision (OpenCV, not to be confused with OpenCL) library allowed us to read in this image data, as well as display the image data to a monitor. The images were read into two-dimensional Numerical Python (numpy) arrays. These numpy arrays could then be passed as data buffers to the OpenCL kernels for parallel processing.

Chapter 2 describes the general steps of the LRF algorithm. In order to realize this system on a GPU, each of the steps of the LRF algorithm was made into an OpenCL kernel. The OpenCL Sobel kernel takes in the input frame and outputs an edge map frame. It was optimized using work-groups and work-items as described above, such that the Sobel convolution windows could quickly operate on a group of pixels by accessing only the local memory. In the FPGA version discussed in Chapter 2, the Sobel calculation was approximated as $|D| = |D_x| + |D_y|$, but the GPU implementation makes use of the optimized hypotenuse (hypot) function to perform

the full Sobel calculation, $|D| = \sqrt{(D_x^2 + D_y^2)}$. Similarly, a local memory optimized general two-dimensional convolution function was created to handle the blur of the edge map and creation of the keep maps discussed in Chapter 2. In the FPGA version, the image filter for this blur convolution was a simple mean filter due it being easy to implement in Verilog. However, using the Python frontend on the host side, the GPU version could easy pass over the Gaussian filter recommended by Aubailly, et al. [15]. Finally, when satisfied that the LRF algorithm was running correctly on the recorded data, we used the Python ctypes library and the frame grabber application program interface (API) to read data from the Camera Link camera into numpy arrays, which could then be GPU processed.

For our experiments, we used the GPUs on the NVIDIA Quadro K5000 and the NVIDIA GeForce GTX 690 video cards. The GPU systems could perform all of the steps of the LRF algorithm and output the result around real-time speeds. That was an encouraging result, so we explored adding additional features to the GPU version that were absent from the FPGA version. The most important of these additional features were variable frame buffer size and image stabilization. Perhaps the most powerful feature of the GPU implementation was the relative ease with which we could implement different sized frame buffers. The GPUs we were testing contained over 4 GB of on-board GDDR5 video memory. Unlike the FPGA, in which the DDR memory was difficult to use (see Chapter 3), the video RAM on the GPU was handled by OpenCL internally. Thus, as long as sufficient memory was available on the card, it was a simple matter to create an adjustable frame buffer parameter to store more or less synthetic frames, as desired. With our 512x512 pixel images, we found that we could create a frame buffer as large as 60 frames or more. However, the larger frame

buffers came at the cost of reduced frame rate, such that a 60 frame buffer of 512x512 images might drop below our minimum real-time speed threshold of 30 frames per second. Therefore, most testing was still done using the original 30 frame buffer.

Another important addition to the LRF system made possible with the GPU host/device dynamic was image stabilization. Image stabilization is compensating for unintended motion in the image. For example, it is often used to correct the shaking of a camera. A large problem with correcting atmospheric turbulence using an algorithm like LRF is the fact that atmospheric turbulence seems to introduce small, random, non-linear apparent motion throughout the image. When comparing one frame to a subsequent frame using LRF, this apparent motion can cause the incorrect pixels to be compared with each other and yield erroneous results. Figure 4.3 shows a visual representation of a random vector field similar to the motion one might expect from turbulence [28].

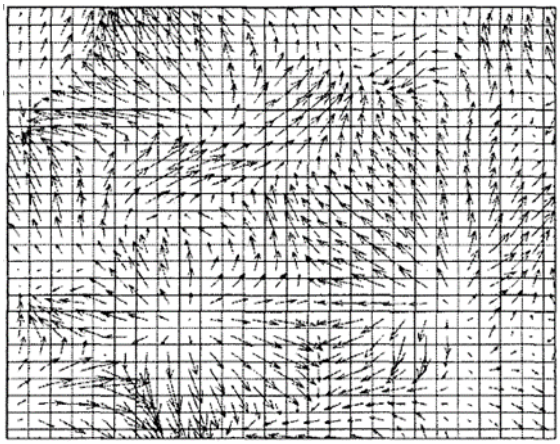


Figure 4.3: Random vector field similar to apparent motions of turbulence. [28]

Image stabilization is typically done on entire images to correct global motion. However, when dealing with atmospheric turbulence, the apparent motion is in different directions throughout the image. Hence, we developed an image stabilization system in which the original image is split into several sub-images, and then image stabilization is applied to each one. Afterward, the sub-images are then reconstructed back into a stabilized image.

The image stabilization method we used is called phase correlation. It is a relatively simple method used to correct for translational motion between images [29]. First, we created a reference frame, which was the mean or median (changeable) of an adjustable number (typically 30) of input frames stored in a reference frame buffer. The 2D Fourier transform is then computed on both the reference frame and the current live input frame that we want to stabilize.

(6)

$$\mathbf{G}_\mathrm{a} = \mathcal{F}\{\mathrm{src}_1\}, \mathbf{G}_\mathrm{b} = \mathcal{F}\{\mathrm{src}_2\}$$

Where src*1* and *src2* are the input and reference images. Next, the cross-power spectrum is calculated:

(7)

$$R = \frac{\mathbf{G}_\mathrm{a} \circ \mathbf{G}_\mathrm{b}^*}{\left|\mathbf{G}_\mathrm{a} \circ \mathbf{G}_\mathrm{b}^*\right|}$$

Where in the above equation, ∘ denotes that the transforms are multiplied element-wise, and * is the complex conjugate. We then find the normalized cross-correlation with the inverse Fourier transform:

(8)

$$r = \mathcal{F}^{-1}\{R\}$$

And finally, the location of the peak is given by:

(9)

$$(\Delta x, \Delta y) = \text{argmax}_{(x,y)}\{r\}$$

This peak represents the translational shift between the input frame and the reference frame [29]. This shift is then used to remap the input frame pixel locations to the corrected pixel locations, canceling the shift and stabilizing the image. The stabilized image was then sent through the LRF algorithm as normal.

Testing this image stabilization concept was initially done on the host side of the system using the OpenCV phase correlate function on a variable number of sub-images between 4 (each sub-image 128x128 pixels) and 32 (each sub-image 16x16 pixels). Larger numbers of sub-images would typically yield more accurate results, but at the cost of output frame rate. While this CPU processing was much slower than the rest of the GPU pipeline, it was useful for determining whether or not the stabilization made enough of a difference in image quality to attempt to implement it on the GPU.

4.3 Promising Results

The GPU parallel hardware implementation of the LRF algorithm produced comparable – and in some cases, superior – results to the FPGA implementation in a fraction of the development time. Both the FPGA and GPU versions feature an adjustable LRF blur radius. The GPU version, however, also allows for an adjustable synthetic frame buffer size, changeable blur method (either mean or Gaussian), and adjustable keep map averaging kernel radius. Both systems can process either live camera data or recorded video. The FPGA version using the IIR filter method to approximate a circular frame buffer (see Chapter 2) was capable of very fast frame rates on the order of 150 FPS with 512x512 resolution. The GPU version, on the other hand, is capable of considerably improved image quality over the FPGA version due to image stabilization, a potentially larger frame buffer, and use of a Gaussian blue for

the LRF calculations. Currently, the GPU version operating on 512x512 resolution with a 30 synthetic frame buffer reaches frame rates of 40 to 50 FPS on a NVIDIA GeForce GTX 690 graphics card. Future work will involve further optimizing the GPU implementation by updating the LRF kernels to utilize local memory, and be converting the image stabilization to the GPU. These improvement should result in even faster frame rates for the GPU system. The Figures below present images of the results of the GPU implementation.

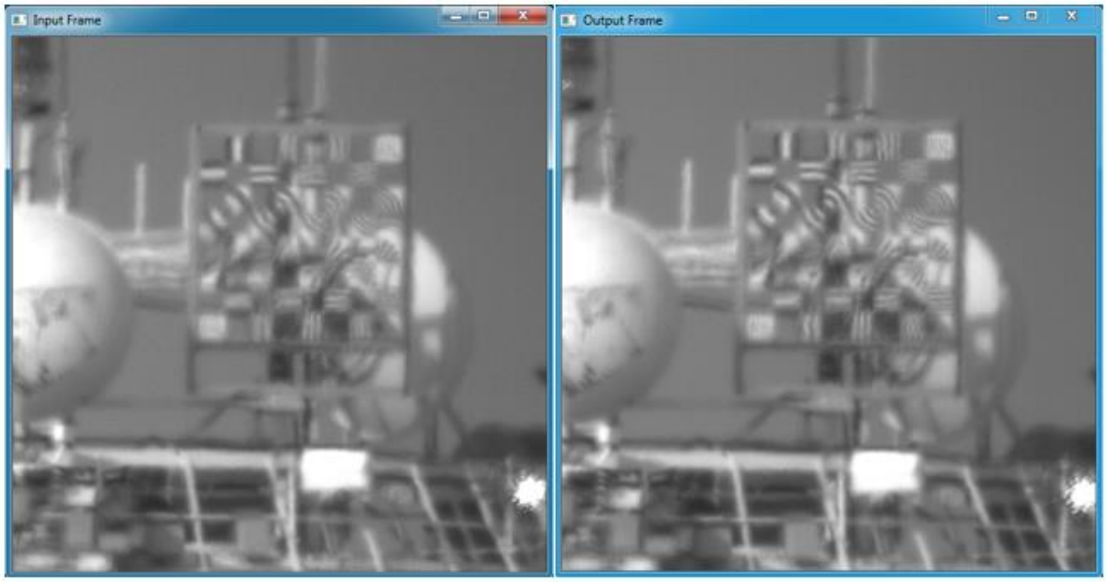


Figure 4.4: GPU LRF Results under moderate turbulence conditions. (a) (*left*) Live input video, unprocessed. (b) (*right*) LRF output from GPU. The lines on the distance resolution chart are noticeably sharper. The “IOL” in the upper right corner of the chart is more legible.

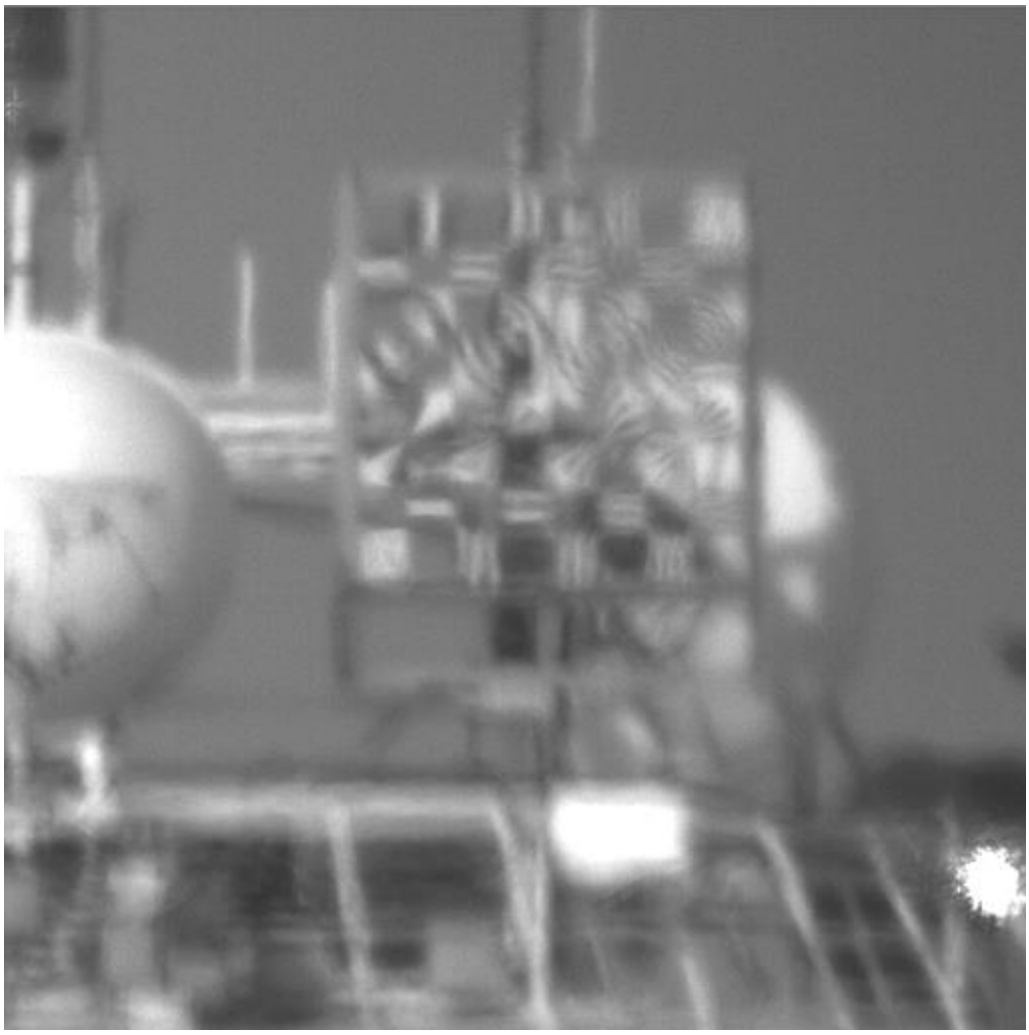


Figure 4.5: Live, unprocessed video data under poor turbulence conditions. Most of the lines on the long distance resolution chart are blurred, and none of the letters are readily legible.

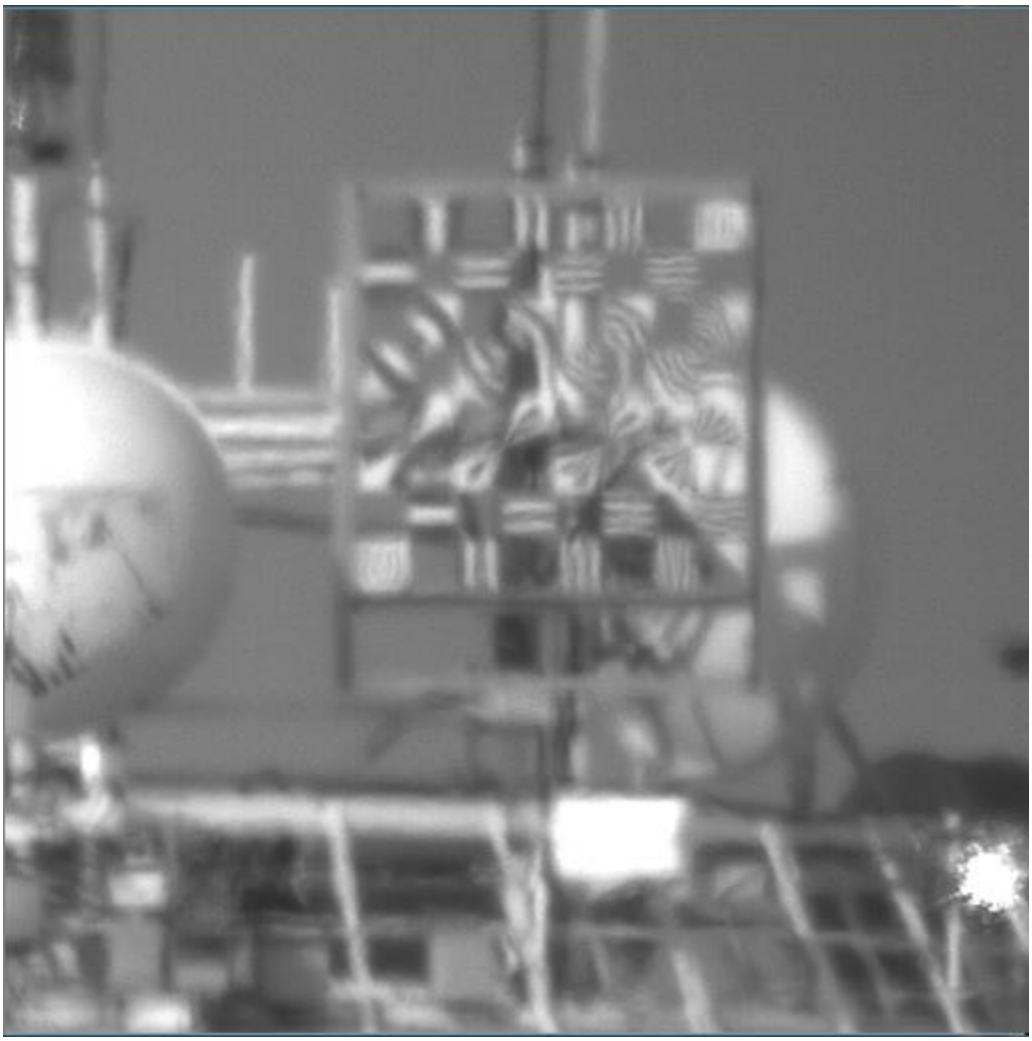


Figure 4.6: LRF output after FPGA processing of the image in Figure 4.5. The approximated frame rate solution described in Section 3.1 was used with keep value 2. The LRF radius for the mean blur filter was 7.

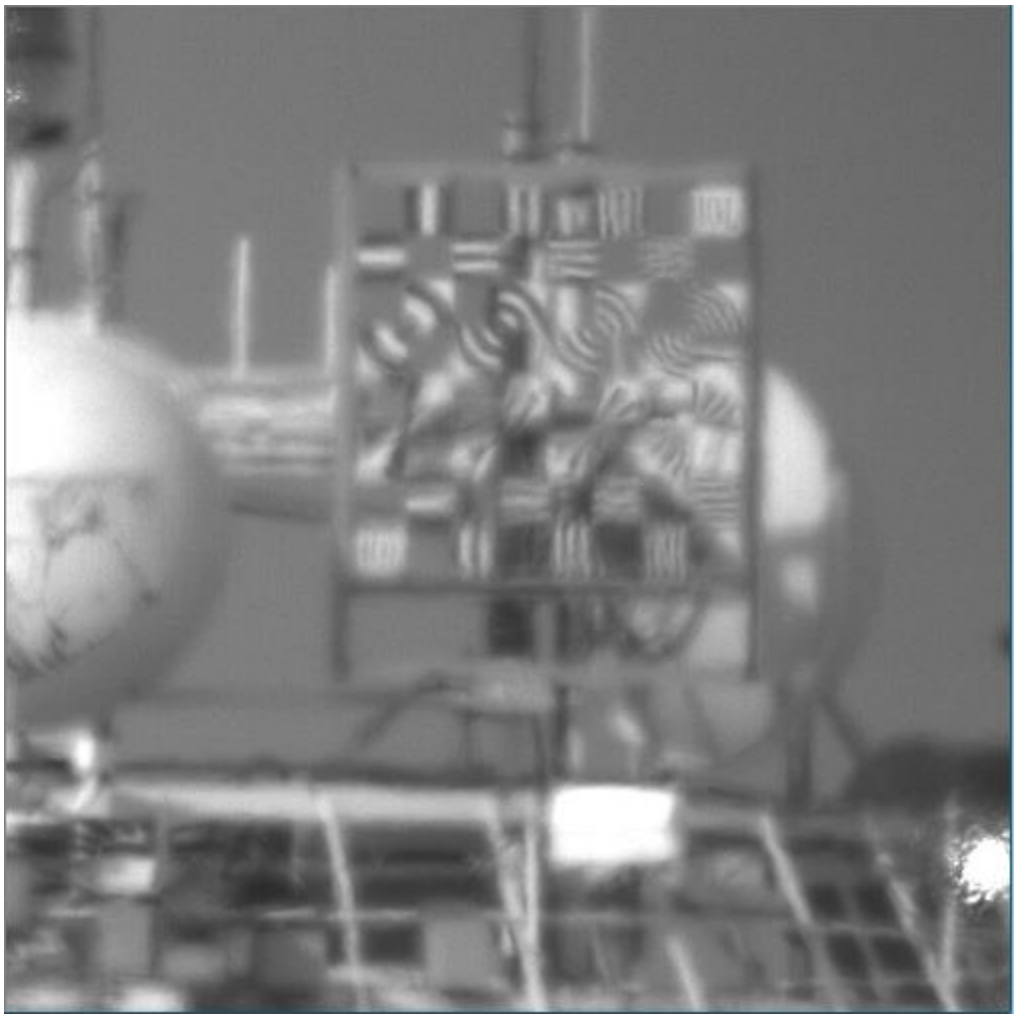


Figure 4.7: GPU output after processing of the image in Figure 4.5. Image stabilization using a mean reference frame made from a buffer of 30 frames.16 sub-images were used in the image stabilization. A circular frame buffer of 30 frames was used for the LRF calculations. A Gaussian blur was utilized with an LRF radius of 4 and standard deviation of 1.5.

Inspection of Figures 4.5, 4.6, and 4.7 above clearly demonstrates the atmospheric turbulence mitigation of LRF. The GPU results are of comparable – and perhaps even superior – quality to the FPGA results, judging by the sharpness and legibility of the two long distance resolution charts.

Chapter 5

CONCLUSION

5.1 Conclusion

An overview of the LRF algorithm was presented. The first and second generations of a FPGA hardware implementation of LRF were described in detail. An in-depth overview of the process and methodology of building upon the second generation system was discussed in Chapter 3. An alternative approach to LRF hardware acceleration using a GPU was then explored in Chapter 4. LRF was originally implemented in software as a post-processing technique, but the relative simplicity of the algorithm and the large amounts of data to be processed suggested the adoption of a parallelized hardware approach. This concept was first proven to be viable with the first generation system, which was capable of LRF processing in real time, albeit with some limitations. The second generation system addressed each of the limitations of the first system, as well as adding additional features for testing and simulation. However, fully realizing the second generation system proved to require too much development time to be viable using the current development platforms. An alternative approach using a GPU was developed with comparable atmospheric turbulence mitigation and real-time frame rate capabilities to the FPGA version.

5.2 Future Work

Future work involves fully integrating image stabilization onto the GPU so that all of the image processing is done in parallel. Another significant enhancement would be to separate the blur do to object motion from the blur due to atmospheric turbulence so that the system could properly process video with moving targets. Optimizing the system to process data at extremely high frame rates is also desirable. Higher

resolution, such as 1k x 1k or 2k x 2k, processing while maintaining real-time LRF processing is another potential improvement. Presently, the LRF algorithm has only been used with greyscale cameras. A future improvement will be to update the system so that it will operate with a color camera. Future versions will also expand the input and output capabilities of the system so that it is compatible with other interfaces than Camera Link, such as DVI or HDMI.

REFERENCES

1. Baumgartner, Dustin D., and Bruce J. Schachter. “Improving FLIR ATR performance in a turbulent atmosphere with a moving platform.” In Proc. of SPIE Vol, vol. 8391, pp. 839103-1. (2012).

2. Maignan, William, David Koeplinger, Mathieu Aubailly, Gary W. Carhart, Fouad Kiamilev, and J. Jiang Liu. "Hardware acceleration of lucky-region fusion (LRF) algorithm for image acquisition and processing ", Proc. SPIE 8720, Photonic Applications for Aerospace, Commercial, and Harsh Environments IV, 87200B (2013); doi:10.1117/12.2016341

3. van Eekeren, Adam WM, Klamer Schutte, Judith Dijk, Piet BW Schwering, Miranda van Iersel, and Niek J. Doelman. “Turbulence compensation: an overview.” In SPIE Defense, Security, and Sensing, pp. 83550Q-83550Q. International Society for Optics and Photonics, (2012).

4. Oreifej, Omar, Xin Li, and Mubarak Shah. “Simultaneous video stabilization and moving object detection in turbulence.” (2013): 1-1.

5. Zhu, Xiang, Peyman Milanfar. “Removing atmospheric turbulence via space-invariant deconvolution.” (2013): 1-1.

6. Zhu, Xiang, and Peyman Milanfar. “Image reconstruction from videos distorted by atmospheric turbulence.” In IS&T/SPIE Electronic Imaging, pp. 75430S-75430S. International Society for Optics and Photonics, (2010).

7. Aubailly, Mathieu, Mikhail A. Vorontsov, Gary W. Carhart, and Michael T. Valley. “Automated video enhancement from a stream of atmospherically-distorted images: the lucky-region fusion approach.” In SPIE Optical Engineering+ Applications, pp. 74630C-74630C. International Society for Optics and Photonics, 2009.

8. Fried D. L., “Anisoplanatism in adaptive optics,” JOSA A, vol. 72, pp. 52–61 (1982).

9. Baldwin, J. E., Tubbs, R. N., Cox, G. C., Mackay, C. D., Wilson, R. W. and Andersen, M. I., "Diffraction-limited 800 nm imaging with the 2.56 m Nordic Optical Telescope," Astron. and Astrop., 368, L1-L4 (2001).

10. Fried, D. L., "Probability of getting a lucky short-exposure image through turbulence," J. Opt. Soc. Am. A, 68, 1651-1658 (1978).

11. Carhart, G. W., and M. A. Vorontsov. “Synthetic imaging: nonadaptive anisoplanatic image correction in atmospheric turbulence.” Optics letters 23, no. 10 (1998): 745-747.

12. Vorontsov, Mikhail A., and Gary W. Carhart. “Anisoplanatic imaging through turbulent media: image recovery by local information fusion from a set of short-exposure images.” JOSA A 18, no. 6 (2001): 1312-1324.

13. Vorontsov, Mikhail A. “Parallel image processing based on an evolution equation with anisotropic gain: integrated optoelectronic architectures.” JOSA A 16, no. 7 (1999): 1623-1637.

14. Droege, Douglas R., Russel C. Hardie, Brian S. Allen, Alexander J. Dapore, Jon C. Blevins. "A real-time atmospheric turbulence mitigation and super-resolution solution for infrared imaging systems." Proc. SPIE 8355, Infrared Imaging Systems: Design, Analysis, Modeling, and Testing XXIII, 83550R (May 18, 2012); doi:10.1117/12.920323

15. Aubailly, Mathieu, M. Vorontsov, G. Carhart, and M. Valley, "Video Enhancement through Automated Lucky-Region Fusion from a Stream of Atmospherically-Distorted Images," in Frontiers in Optics 2009/Laser Science XXV/Fall 2009 OSA Optics & Photonics Technical Digest, OSA Technical Digest (CD) (Optical Society of America, 2009), paper CThC3.

16. Muller, R. A. and A. Buffington, "Real-time correction of atmospherically degraded telescope images through image sharpening," J. Opt. Soc. Am. 64, 1200–1210 (1974).

17. Danielsson, P.E., Seger, O. "Generalized and Separable Sobel Operators" in "Machine vision for three-dimensional scenes", Herbert Freeman (ed), Academic Press (1990)

18. Gonzalez, Rafael C. and Richard E. Woods. *Digital Image Processing*. Pearson Education Inc, 2008. pp. 157-168.

19. Baptista, Rafael. “Fast Approximate Distance Functions,” Flipcode, 27 June 2003. http://www.flipcode.com/archives/Fast_Approximate_Distance_Functions.shtml (16 December 2013)

20. Ashenden, Peter. *Digital Design An Embedded Systems Approach Using VHDL*. Burlington: Morgan Kaufmann, 2008. pp. 400-425.

21. Xilinx. 7 Series FPGAs Memory Interface Solutions v2.0 User Guide. 19 June 2013. pp. 145-163.

22. Xilinx. VC709 Evaluation Board for the Virtex-7 FPGA User Guide. 4 June 2013. pp. 5-66.

23. Toyon. *Boccaccio – FMC Camera Link*. ISR Algorithms. http://www.toyon.com/products/boccaccio (January 2013)

24. PULNiX America, Inc. Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers. October 2001. pp 3-1 – 5-3.

25. Xilinx. "7 Series FPGAs Overview," 17 December 2014. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (12 January 2015)

26. Tsuchiyama, Ryoji, T. Nakamura, T. Iizuka, A. Asahara, J. Son, and S. Miki. *The OpenCL Programming Book*. Fixstars 2010. http://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/what-is-opencl/ (27 December 2014)

27. Scarpino, Matthew. *OpenCL in Action: How to Accelerate Graphics and Computation*. Shelter Island, NY: Manning Publications Co., 2012. pp 43-93.

28. Frakes, David H., J. Monoco, and M. J. T. Smith, "Suppression of Atmospheric Turbulence in Video Using an Adaptive Control Grid Interpolation Approach." In Acoustics, Speech, and Signal Processing, pp. 1881-1884 vol 3. IEEE International Conference, (2001)

29. OpenCV Dev Team. "Motion Analysis and Tracking." OpenCV 3.0.0-dev Documentation, 21 April 2014. http://docs.opencv.org/trunk/modules/imgproc/doc/motion_analysis_and_object_tracking.html#phasecorrelate (21 November 2014)

Appendix

PERMISSION TO REPRINT COPYRIGHT MATERIAL

A.1 Request for Permission

Christopher Jackson

114 Evans Hall

Newark, DE 19716

302-353-8922

chrisrj@udel.edu

Manning Publications Co.

20 Baldwin Road

PO Box 261

Shelter Island, NY 11964

Dear Copyright Permissions Department:

I am writing to request permission to reprint a portion of the following work:

Matthew Scarpino

OpenCL in Action: How to Accelerate Graphics and Computation

ISBN: 9781617290176

Manning Publishing Co.

Copyright 2012

Figure 3.9 on page 66 and Figure 4.6 on page 87.

The request is for permission to include the above content to incorporate into a thesis in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering at the University of Delaware for the Spring 2015 semester.

Jackson, Christopher R. "Real Time Mitigation of Atmospheric Turbulence in Long Distance Imaging Using the Lucky Region Fusion Algorithm with FPGA and GPU Hardware Acceleration." University Microfilms, Inc. 2015

Please be aware that this thesis will be published through the University Microfilms, Inc. (UMI) doctoral dissertation program, and I request that you grant UMI authorization for publication of my manuscript in its entirety.

I believe that your company is currently the holder of the copyright, because the original work states that the copyright is held in Manning Publications Co. as of 2012. If you do not currently hold the rights, please provide me with any information that can help me contact the proper rights holder. Otherwise, your permission confirms that you hold the right to grant this permission.

This request is for non-exclusive, irrevocable, and royalty-free permission. It is not intended to interfere with other uses of the same work by you. I would be pleased to include a full citation of your work and other acknowledgement as you might request.

I would greatly appreciate your permission. If you require any additional information, please do not hesitate to contact me at the email address or number provided above. I would be happy to provide you with a complete current draft of the thesis if you deem it necessary.

Sincerely,

Christopher Jackson
Research Assistant
Electrical and Computer Engineering
302-353-8922
chrisrj@udel.edu

A.2 Permission Granted

As long as the material is properly cited you have permission to use this material.

Thanks
Ted
Customer Support
Manning Publications Co.