Computer Vision for Light-Field Medical Imaging: FPGA Image Processing System and Depth Acquisition GAN

Colin Pate

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington
2019

Committee:

Joshua R. Smith

Brian Curless

Linda Shapiro

Program Authorized to Offer Degree:
Electrical Engineering

University of Washington

Abstract

Computer Vision for Light-Field Medical Imaging: FPGA Image Processing System and Depth Acquisition GAN

Colin Pate

Chair of the Supervisory Committee:
Professor Joshua R. Smith
Allen School of Computer Science and Engineering,
Department of Electrical Engineering

Light field imaging demands a higher data bandwidth and more post-processing than traditional photographic imaging, a characteristic that is compounded in real-time medical applications. Rendering a realistic sub-aperture image captured from a light field array requires a precise 3D understanding of the scene being imaged. For use in a surgical environment, these images must be captured and rendered with consistently high resolution and low latency. In this paper, we present two systems to improve the latency and precision of the Proprio Vision medical light field array. The first of these improvements is a hardware and software system that implements an FPGA as a modular platform to pre-process and intelligently downsample images sent from the image sensors before they are processed by the main PC in the array, reducing the workload of the main GPU. The second proposed system is a neural network that generates depth maps from light field images. This system reduces depth capture time and rendering latency by replacing the current structured-light depth capture solution. The model is implemented as a Generative Adversarial Network to improve the accuracy and realism of the generated depth maps. Together, the two proposed systems reduce the hardware cost and complexity of the Proprio Vision array while improving latency and image quality.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to extend my gratitude to my advisor, Prof. Joshua Smith, for connecting me with Proprio and giving me the opportunity to work on this research. I would also like to thank everyone at Proprio, especially lead engineer James Youngquist, for supporting me in my work and allowing me to research for my thesis while experiencing the startup environment.

I am also grateful to Saman Naderiparizi for providing advice and assistance when I was designing the FPGA to image sensor interface and high-speed PCB.

Chapter 1

INTRODUCTION

The application of augmented reality in a surgical setting provides a unique set of engineering challenges and opportunities for innovation. High-resolution imaging and 3D visual cues can provide surgeons with improved sensing abilities and enable them to perform with enhanced precision and accuracy. However, the spatial and temporal error of such a system must be held to extremely low tolerances to ensure safe outcomes [1]. One of the imaging systems used in this application is a 12-camera light field array developed by Proprio Vision. This array is designed to generate real-time light field images that allow the surgeon and observers to zoom and reposition the surgical scene freely while operating, with 3D data such as CAT scans overlaid. A visible light projector in the array projects structured light patterns which are sensed by the cameras and used to acquire the 3D geometry of the scene being rendered. This 3D geometry is used to generate the light field sub-aperture images (SAI) for VR rendering. We make the following contributions to the light field array:

- An FPGA image processing system to pre-process images from camera image sensors
- A machine-learning geometry acquisition algorithm to obviate the need for structure light

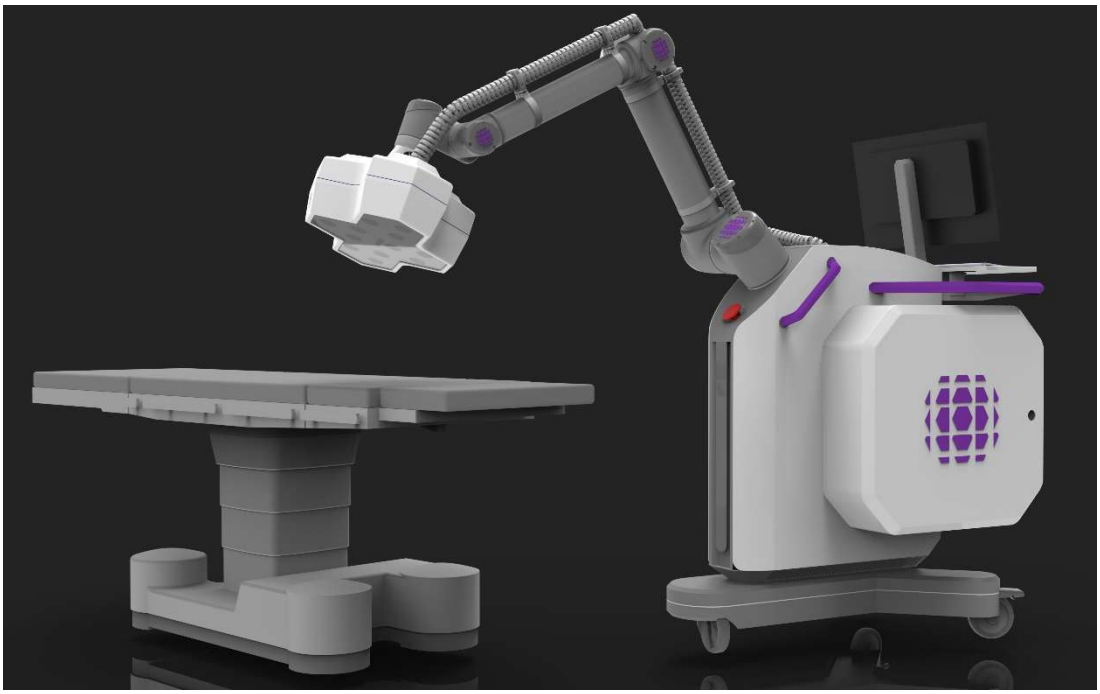

Figure 1.1: A render of the Proprio light field array mounted on the robotic arm and cart, alongside a surgical table. The cart contains the robotic arm controller and the PC that renders the VR images.

This thesis will describe the design details of both systems and how they lower the cost and increase the utility of the light field imaging system.

1.1 Background

In this section, the current light field imaging pipeline is explained and compared with other methods, and the new imaging systems are proposed as alternatives.

1.1.1 Light Field Imaging

Light field imaging captures the intensity, color, and spatial location of light rays within a scene [2]. The combination of intensity data and spatial information can be used to generate a 3D representation of a scene with directional color and intensity data available for each point. This

Figure 1.2: A view from underneath the array, showing the cameras, structured light projector, and a surgeon wearing the head mounted display.

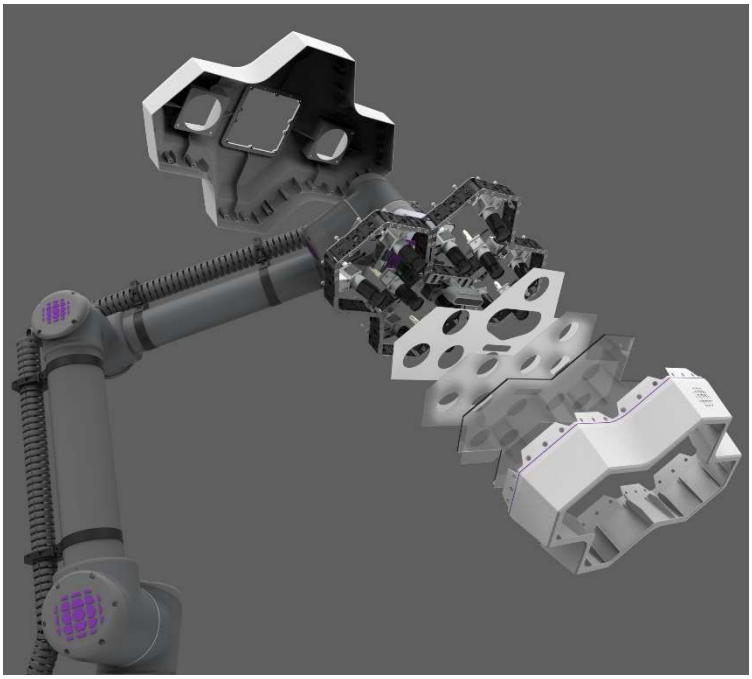

Figure 1.3: A render of an exploded view of the array.

process is similar to ray tracing in computer generated imagery, where rays from the virtual camera are traced to their originating light source. However, in light field imaging, the rays are traced to the nearest real camera, preserving specularity and directional variations in appearance. This makes it possible to generate a realistic image from a 'virtual camera' placed anywhere around the scene, also known as a sub-aperture image (SAI). Viewing a light field image in a virtual reality headset requires the rendering of two SAIs, one for each eye.

In the Proprio Vision Light-Field System (LFS), rays from the cameras nearest to the spatial position of the viewer are used to render the SAIs, resulting in photorealistic renders that capture the specularity of surfaces in the scene.

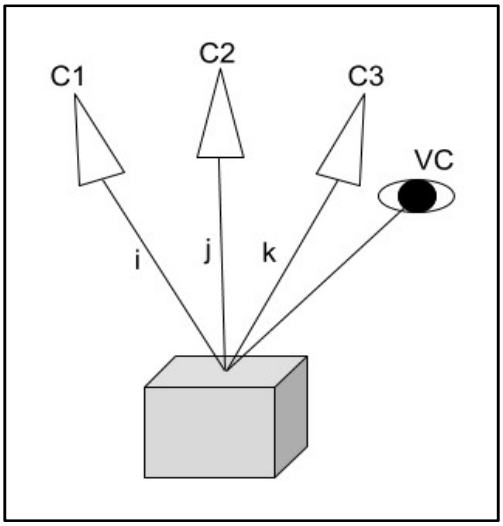

Figure 1.4: This figure depicts the geometric mapping of light field rays to 3D objects. C1, C2, and C3 represent real cameras imaging an object. Lines i, j, and k, denote light rays in the cameras fields of view, all coincident at the same point on the object. VC represents the virtual camera for which a SAI should be generated. The color and intensity from k will have the most influence on this pixel in the SAI because C3 is closest to the position of the virtual camera.

1.1.2 Data Pipeline

The LFS currently uses 12 4-megapixel USB 3.0 cameras connected to a host PC to capture color images. The host PC communicates and powers the cameras through USB 3.0 cards connected to the system PCIe bus. Images from the cameras are buffered in the system DRAM, from which they are copied to the GPU for light field rendering or geometry acquisition. The USB cameras can capture and transmit 2048 × 2048 Bayer-pattern images at approximately 75fps, limited by USB bandwidth. For each rendered frame, images from every camera must be converted from Bayer to RGB on the GPU and warped to correct for lens distortions. Each image takes 0.85ms to demosaic and 0.34ms to remap on a high-end workstation GPU. The minimum frame rate that VR imagery can be rendered without causing nausea or viewer discomfort is around 60fps [3]. At this frame rate, image pre-processing uses 85.6% of the available GPU processing time, leaving very little left over for SAI rendering. Offloading these operations to an external hardware accelerator would relieve this bottleneck, allowing faster and higher quality light field rendering.

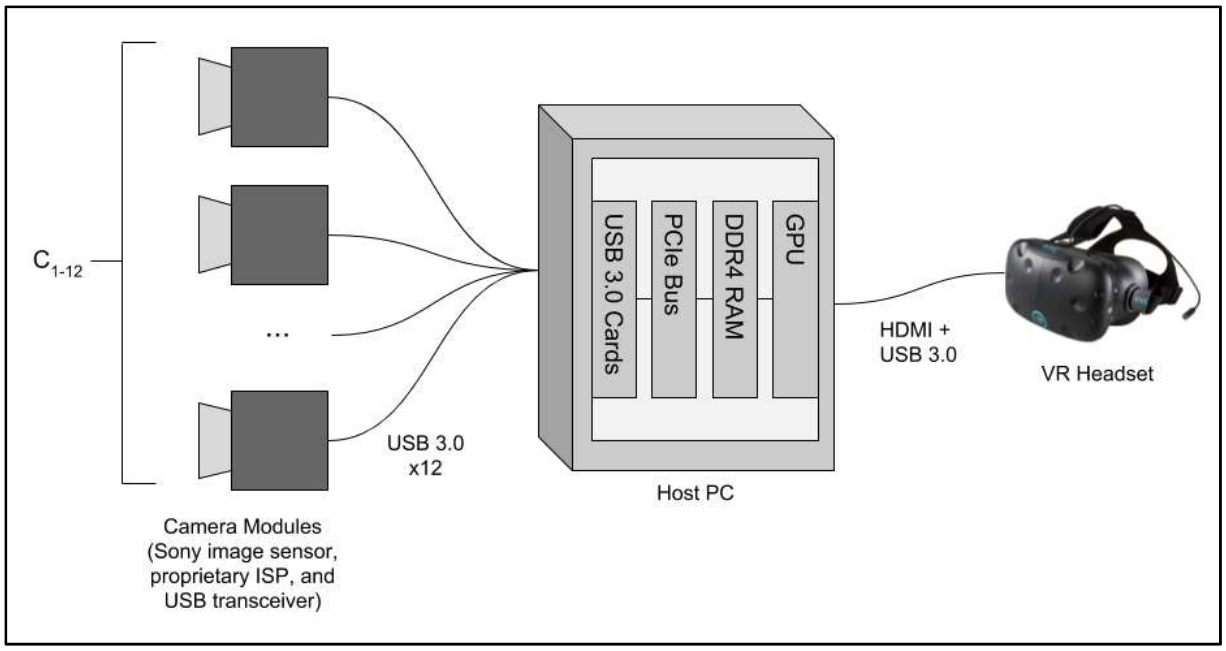

Figure 1.5: The data pipeline used for light field data capture and rendering. Uncompressed Bayer-pattern images are received through USB 3.0 cards from 12 cameras. These images are buffered in the system DDR4 and manipulated by the GPU for rendering in the VR headset.

1.1.3 Geometry Capture

A 1080p consumer HD projector is used to project temporal gray code structured light patterns for geometry capture, using the algorithm proposed in [4]. Gray codes allow the depth capture algorithm to localize 3D points by determining whether their intensity changes between subsequent frames. The projector is driven by a Raspberry Pi single-board PC that also drives the camera trigger inputs. This allows the camera digital shutters to be precisely triggered with the same timing as the projected gray code images. Gray codes are projected separately for depth capture in the X and Y directions, with spatial resolution doubling for every 2 gray codes. Thus to

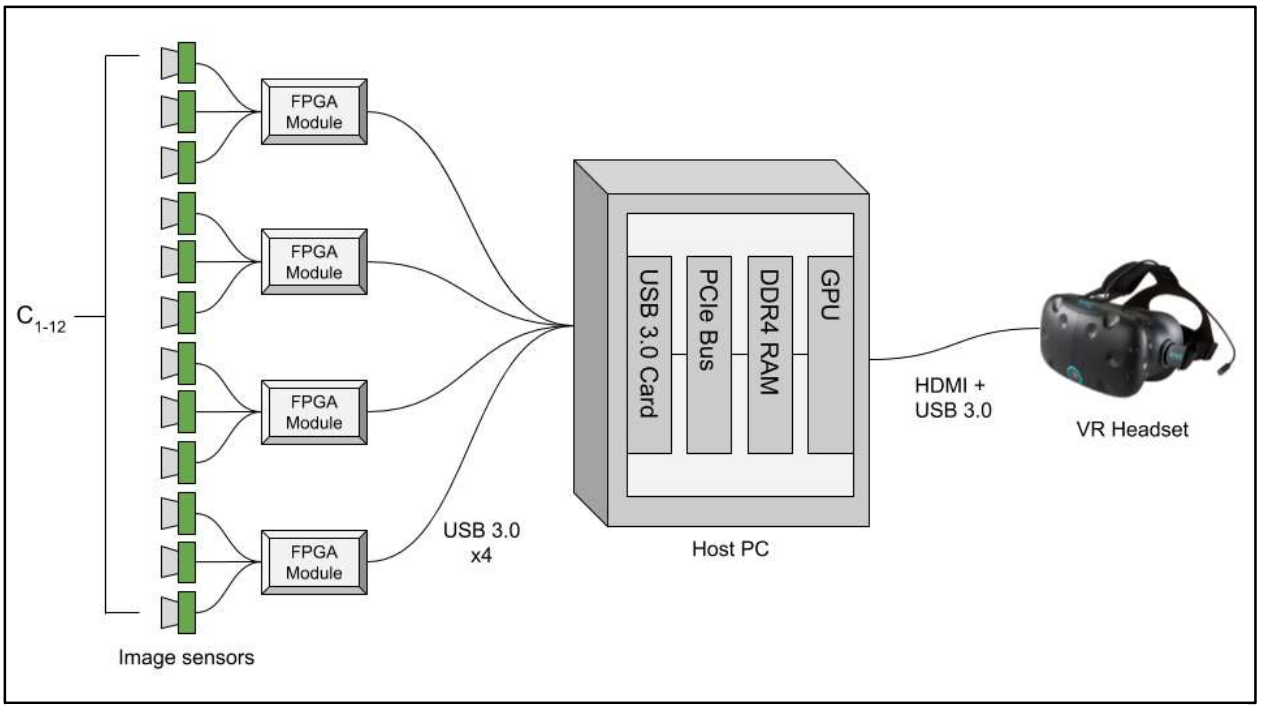

Figure 1.6. The proposed new data pipeline for light field rendering. FPGA modules communicate directly to image sensors and receive raw Bayer-pattern images. These images are processed by the FPGAs and sent fully or in part to the host PC when requested.

generate a depth map of dimensions X × Y, the structured light capture requires 2(log2(X) + log2(Y)) gray coded images. At a resolution of 512 × 512, this takes approximately 720ms in total and results in a depth map with a depth accuracy we measured at about 0.1mm. The spatial position of each point must be agreed upon by at least 3 cameras in order to eliminate spurious matches. Most of the gray code points are pruned through this process due to artifacts such as reflections and thin structures, resulting in incomplete depth maps that must be meshed to create a coherent structure. This necessitates a compromise between accuracy and mesh completeness. A more complete mesh may be less accurate, as artificial geometry inserted in unknown areas may not reflect the true geometry of the scene being captured. Other existing depth imaging systems such

as Intel RealSense and Microsoft Kinect use static structured light patterns to capture depth at up to 90fps and 1MP, but with decreased spatial accuracy [5][6].

1.2 Project Overview

1.2.1 FPGA Image Processing System

The first proposed system is the FPGA in-array image processing system. This system, shown in Figure 1.4 will replace the current integrated USB camera modules with an FPGA SoM (System on Module) connected to one or more image sensor modules. The FPGAs will control the image sensor modules and process images before they are sent to the host PC through USB 3.0. The specific improvements this system provides are:

1. On-array demosaicing. The FPGA handles the conversion of Bayer pattern images to RGB color images, removing this workload from the GPU to allow faster light field rendering.
2. On-array lens correction. The FPGA remaps images from the image sensors as they are received, reversing lens distortion so all images match the standard pinhole projection and removing this workload from the GPU to allow faster light field rendering.
3. Region-of-interest selection. The host PC can select which regions to receive from each camera, resulting in reduced interface and memory bandwidth requirements.
4. Block matching. Block matching is implemented on the FPGA to provide a rough depth map of the scene for geometry refinement or pre-conditioning.
5. Reduced tether size and weight. The tether from the array to the host PC currently contains 12 USB 3.0 cables, along with power and networking. Connecting 3 cameras to a single FPGA module could reduce this to 4 cables.

6. Reduced cost. The current camera modules cost $944 each, with a total cost of $11328. Equivalent image sensors cost $250 each and FPGA SOMs that can be connected to 3 cameras cost $480 each, with a total cost of $4440. Replacing these with image sensor modules and FPGA SOMs will save $6888 per array.

1.2.2 Machine Learning Geometry Acquisition

The second proposed system is the geometry acquisition neural network. This algorithm uses a CNN (Convolutional Neural Network) implemented in a GAN (Generative Adversarial Network) architecture to produce depth maps in real time without the need for structured light. The advantages of the ML depth algorithm include:

1. Faster scan time. Unlike the structured light system, the Depth GAN only requires one frame from the array and takes less than 100ms to infer depth.
2. No need for a projector. The Depth GAN does not require a structured light projector, resulting in decreased cost, weight, and complexity of the camera array.
3. Complete depth maps. The structured light provides accurate depth maps by pruning points where fewer than 3 cameras agree on the depth. This results in holes in the depth maps at sharp discontinuities, thin structures, and highly reflective surfaces. The Depth GAN uses global cues from the full image to produce a complete depth map and infer depth in regions where isolated measurements may not produce accurate data.

1.3 Associated Publications

1.3.1 FPGA Image Processing

FPGAs are a common platform for low-level image processing due to their ability to process large amounts of data with low latency and power consumption [7]. They are most often used as a layer between the main computing device and the image sensor. Image sensors frequently use low-level serial protocols such as MIPI CSI-2 for image transmission. An FPGA can convert from this protocol to a protocol accepted by a standard PC, such as USB or PCIe. If the image sensor features a built-in image signal processor (ISP), it can be supplemented or replaced with an FPGA to provide more processing algorithms and flexibility. FPGAs can also be used to perform demosaicing and lens-correction, functions that require a large amount of memory bandwidth on a traditional CPU or GPU but can be implemented in on-chip memory on an FPGA. In some cases, FPGAs have been used for higher-level algorithms such as stereo block matching.

1.3.2 Protocol Conversion

MIPI CSI-2 is the most common transmission protocol used by image sensors and is supported by nearly all sensors available today. While many embedded devices today include CSI-2 receivers, most of them are designed for a very limited range of sensors. Full-sized PCs generally do not support CSI-2 natively. FPGAs, however, feature a large number of flexible high-speed digital I/O ports. Lattice offers low-cost, low-power FPGAs with built in CSI-2 PHYs. Both Xilinx and Altera have published application notes showing how to implement a CSI-2 PHY with standard LVDS RX pins [8]. While the MIPI specification is intended to be available only to members of the

organization, the specification is available online and an open-source implementation for Xilinx FPGAs is available on GitHub [9].

1.3.3 Demosaicing

The conversion from Bayer images to RGB has been tackled in a wide variety of ways, from simple nearest-neighbor averaging to machined learned algorithms [10]. Many algorithms can be adapted for efficient FPGA processing. Rani et al. [11] use a local averaging algorithm to demosaic small images. Bailey et al. [12] implement a much more complex edge-aware interpolation filter with classification, resulting in a reduction of interpolation artifacts.

1.3.4 Image Remapping

The standard transformation matrices for conversion from world coordinates to camera coordinates require that the camera adheres to the pinhole model. This presents a problem when using real-world optics, which introduce distortion to the captured image. This can be corrected by characterizing the lens distortion using a parametric model and re-mapping the camera image after demosaicing. Image remapping uses a transformation matrix or another algorithm to create a new image in which each pixel is copied from a single point in the input image. To produce an accurate output image, an interpolation scheme such as bilinear interpolation should be used. Bilinear interpolation requires an input pixel to output pixel ratio of 4:1. Without intelligent memory access, this can easily consume the entire memory bandwidth of an FPGA system. Memory bottlenecks when remapping large images can be mitigated by using on-chip memory to buffer lines or blocks from the source image. Griesen, et al. [13] describe an FPGA image processing pipeline capable of remapping stereo 1080 × 1920 × 30fps video streams.

1.3.5 Block Matching

A great deal of research has been put into implementing stereo vision algorithms on many different hardware platforms, and FPGAs are no exception. Block matching, where rectangular patches from one stereo image are matched to patches in the opposing image, has also been implemented in programmable logic for optical flow motion estimation in video compression. Every application requires compromises between logic utilization, clock speed, frame rate, resolution, and error. Griesen et al. [13] use a hierarchical pyramid architecture to produce a low-resolution disparity estimate and progressively refine it. Haublein et al. [14] propose a generic, flexible architecture for experimentation on different hardware and depth map quality requirements. Mazumdar et al. [15] do not use FPGAs for block matching but implement a post-processing bilateral filter that refines and up-samples a block-matched depth map using FPGAs.

1.3.6 Depth Acquisition Neural Networks

Neural networks and other machine learning algorithms have shown great success in a variety of computer vision tasks. The first neural networks to beat conventional computer vision algorithms performed tasks such as classification and segmentation. However, machine learning algorithms have begun to compete with traditional methods for regression, including depth perception. Fanello et al. [16] utilize a stereo pair with a structured light projector to achieve 1.3MP depth capture at 375Hz. They use a random forest approach that starts by classifying structured light points and then switches to regression to achieve a continuous result. Generative adversarial networks, which use an adversarial loss function, have shown impressive results on a variety of image translation tasks and have been successfully applied to depth capture. Lore et al. [17] train

the pix2pix network with the Ford Campus Vision and LiDAR data sets. Both datasets provide RGB-D images from car-mounted image capture units. The pix2pix network was successfully able to predict depth from single RGB frames.

1.3.7 Light Field Depth Capture

Light field camera arrays present the opportunity for much higher depth resolution and accuracy than a typical stereo pair, due to their unique architecture and image redundancy. The large number of unique views available allows stereo matching between many different image pairs or the use of specialized algorithms such as depth from defocus. Tao et al. [18] combine cues from depth from correspondence and depth from defocus to acquire a high-resolution depth map from a Lytro light field camera. The Lytro camera uses a micro-lens array with very small baselines, differentiating it from the hardware used for this thesis which has 12 distinct cameras placed at much larger distances apart than the Lytro lenses. Yan et al. [19] use a generative neural network with a pair of images from Lytro cameras to generate high-resolution depth maps.

Chapter 2

FPGA

The first system described in this work is the FPGA image processing device. This chapter will explain the design and functionality of the FPGA system.

2.1 Hardware

2.1.1 Specification

The primary functional requirements for the FPGA system are:

1. Process images of at least 5MP resolution at a minimum of 20FPS, with a minimum of one camera
2. Bidirectional communication with a host PC through USB 3.0
3. FPGA SoC functionality (hard processor built in to FPGA) to allow software-based control of image processing modules
4. Ability to transmit only selected regions of received image to host PC
5. Real-time demosaicing with a minimum 5x5 window size
6. Real-time lens correction

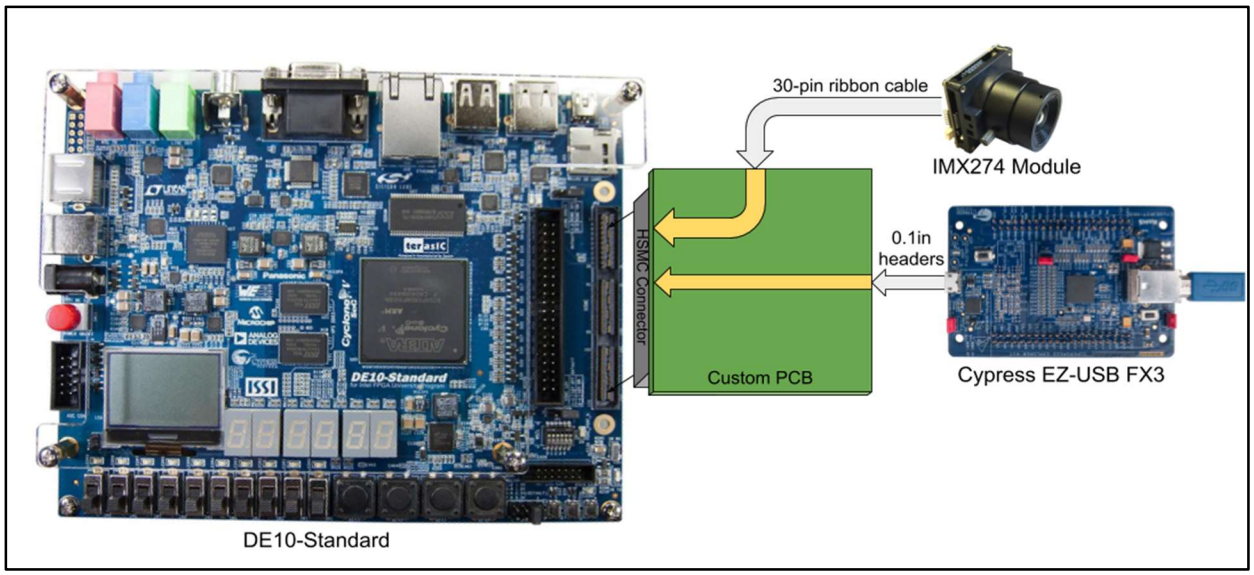

Figure 2.1: Graphical depiction of component connections to custom PCB.

2.1.2 Component Selection

The most important component selections were the FPGA platform and image sensor. The Intel Cyclone V SoC FPGA was chosen due to its low cost, ample logic resources (~110k logic elements on many development boards), and built-in processor. The Terasic DE10-Standard Cyclone V development kit was then selected due to its HSMC (High-Speed Media Connector) support. This interface made it possible to connect both a MIPI CSI-2 camera to the FPGA as well as a USB 3.0 peripheral controller.

The image sensor chosen was the Sony IMX274, selected for its high resolution (8MP at 60FPS), open-source driver availability, and availability as a sensor-on-PCB module. The image sensor was purchased in the LI-IMX274-MIPI-CS module from Leopard Imaging.

The Cypress EZ-USB FX3 Development Kit was used as the Parallel-to-USB translation layer. The FX3 provides an abstraction layer for the USB protocol with a 32-bit bidirectional parallel

interface to the FPGA and a C++ API for PC communication. An adapter from the FX3 to the HSMC connector was purchased and used for testing, however, a custom PCB was designed to allow simultaneous connection of the camera module and FX3, as shown in Figure 2.1.

2.1.3 PCB Design

The DE10-Standard development board features a 180-pin HSMC interface that exposes FPGA pins that can be used as LVDS RX/TX connections or as single-ended digital I/O. This is the only connector on the board that provides high-speed direct connections to the FPGA. Since the camera module and USB FX3 both require high-speed digital I/O, a custom PCB was designed to connect both devices to the FPGA. The schematic from the FX3 HSMC adapter was adapted and modified to expose 4 LVDS RX pairs and 1 LVDS clock RX pair on the HSMC connector for the IMX274's 4-lane CSI-2 interface.

2.2 FPGA Modules

The FPGA image processing pipeline was designed to be as modular and flexible as possible in order to facilitate modification and adaptation for other purposes in the future. Each operation is contained in a SystemVerilog module with sub-modules for atomic processes within the main function.

2.2.1 Full FPGA System Architecture

The FPGA image processing system uses a modular design that is intended to be as flexible and easy to debug as possible. All intermediate images are buffered in the DDR3 SDRAM connected to the SoC HPS (Hard Processor System) through the FPGA-to-SDRAM interface, and each

functional module was created with Intel Avalon-compatible memory mapped or streaming image inputs/outputs. These interfaces are publicly documented and standardized. Each module uses a memory-mapped control interface exposing registers to the ARM CPU that can be accessed directly or through abstraction functions. Moreover, all modules in the block matching system were implemented as Intel System Builder modules, allowing the block matching modules to be connected and parameterized in the Intel System Builder GUI as shown in Figure 2.2. Our intent is to convert all image processing modules to Intel System Builder modules, allowing the full image processing system to be implemented and customized in a graphical interface.

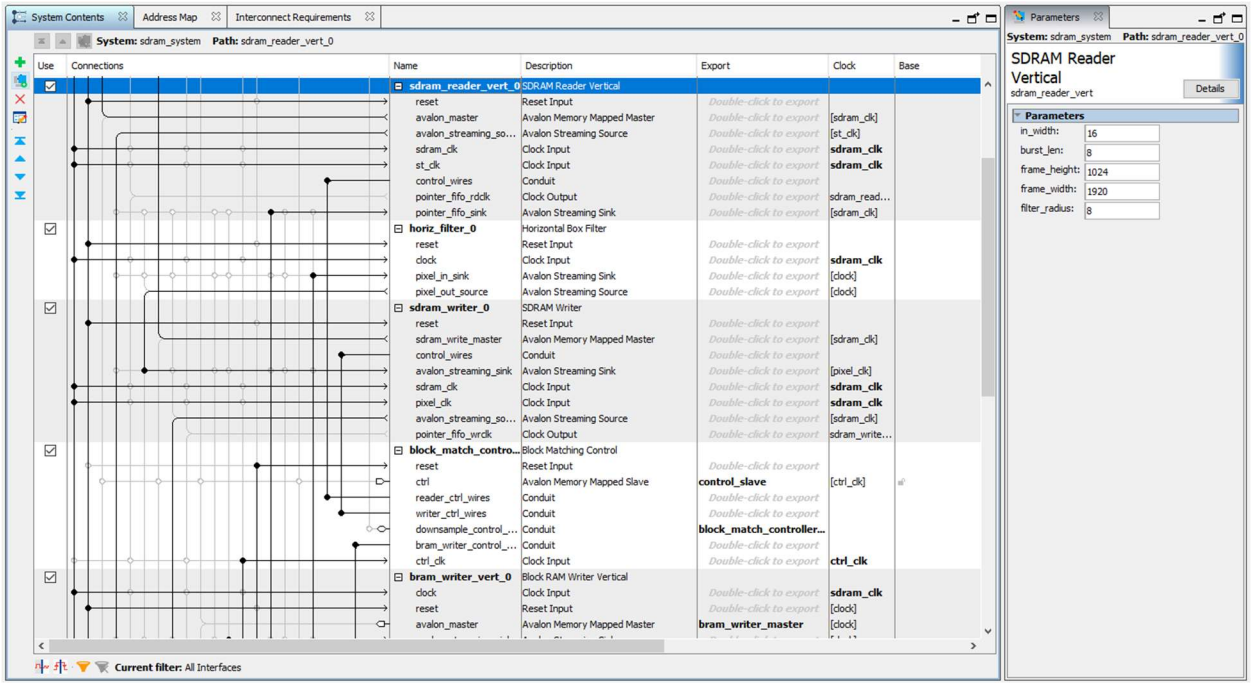

Figure 2.2: Intel System Builder graphical user interface with custom modules.

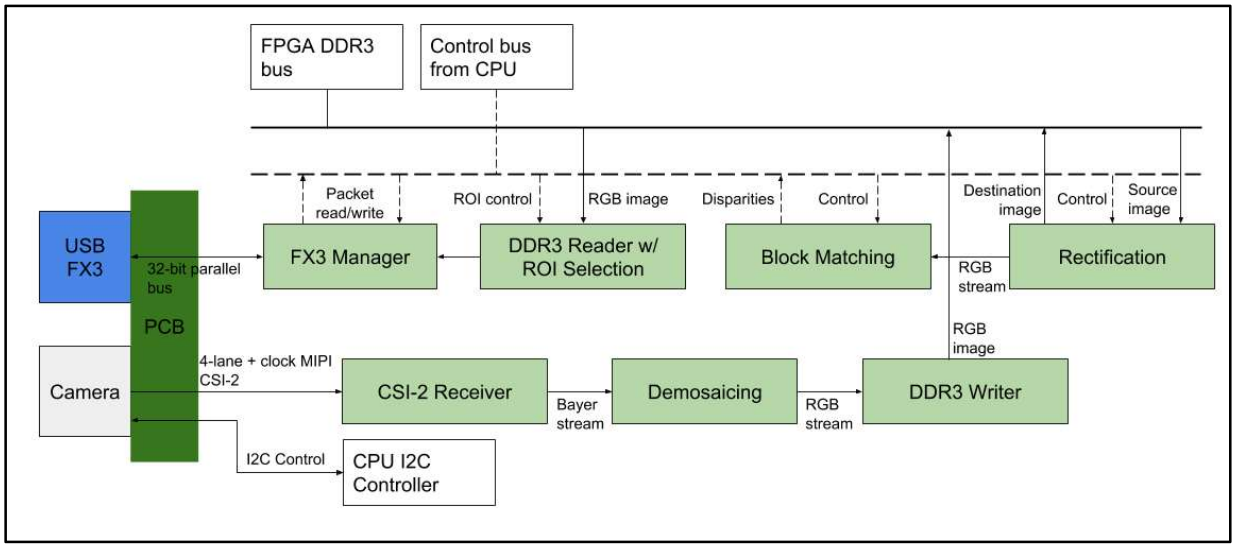

Figure 2.3: Data and control flow diagram of the full FPGA image processing system. All control and data interfaces between modules are Intel Avalon Stream or Memory Mapped interfaces.

2.2.2 MIPI Receiver and Demosaicing

The first stages of the MIPI receiver module are based off David Shah's open-source CSI-2 receiver module [9]. The implementation in [9] was designed for Xilinx FPGAs and written in VHDL, requiring adaptation to SystemVerilog and changes to component instantiations to account for differences in the two FPGA manufacturers' hard IP blocks. After the adapted initial receiver stage, a state machine was implemented to decode packets and convert the interleaved pixels to a pixel stream. This stream is sent to the demosaicing module for conversion to RGB.

The demosaicing module uses 5 row buffers to implement 5 × 5 window-based debayering. The algorithm used is adapted from [20]. This interpolation technique was chosen for its computational simplicity (all operations can be implemented with only bit-shifts and adders) and high PSNR.

After demosaicing, the RGB pixel stream is sent to a DDR3 writer module which buffers the image stream in the DDR3.

2.2.3 Image Transformation

The lens correction image remapping module uses the on-chip memory to buffer image pixel blocks for remapping. This approach greatly reduces random memory accesses compared to a naive remapping solution with no buffering. In a naive solution with bilinear interpolation, the total pixels read is equal to four times the number of written pixels, and the number of random accesses is twice the number of written pixels. For an 8MP 24-bit RGB image of resolution H × W, the total reading and writing time is:

$$t_{rema} = H * W * (t_{seq} + t_{seq} + t_{rand})$$

This corresponds to 176ms per image assuming a 32-bit memory bus with 19ns CAS (t_{rand}) and 1.5ns per sequential read/write (t_{seq}). In a buffered remap solution, the simplest approach is to buffer the image line-by-line in FPGA block memory and read out the remapped image line-by-line. However, this requires that the number of buffered lines be greater than the greatest difference between the min and max source Y coordinate in any line in the destination image. This is feasible for smaller images and small amounts of lens distortion, but for high-resolution images and operations such as rotation it quickly exceeds memory limitations of mid-grade FPGAs. The Cyclone V used in this design has 5.76Mb of block RAM. With 3840 × 24 bit pixel rows in the source image, this allows 62 full lines to be buffered on the FPGA. This would be sufficient for simple lens correction for low-distortion lenses, but we wished to allow for more extreme warping and rotation, the use of multiple cameras, and to conserve FPGA memory for other functions.

The new remapping algorithm buffers source pixels in $W_b \times H_b$ blocks as a compromise between on-chip memory usage and DDR3 memory bandwidth. Figure 2.4 shows the data and control flow diagram of the remapping scheme. Software running on the CPU controls the remapping modules and generates coordinates to map source to destination pixels. A memory-mapped to streaming conversion module is placed at the beginning of the data pipeline. This module reads pixel blocks from the source image and streams them to the pixel block buffer writer. The buffer writer then writes these pixels into 4 rotating memory-mapped buffers (represented as one buffer for simplicity). These buffers are arranged to facilitate bilinear rectification, which requires 4 neighboring source pixels to be read per 1 destination pixel. In even numbered rows of the pixel block, the buffer writer alternates between buffers 0 and 1, alternating for each pixel. The buffer writer alternates between buffers 2 and 3 on odd rows. With this access pattern, every destination pixel can be interpolated using one pixel from each of the four buffers, allowing each destination pixel to be read and interpolated in a single clock cycle.

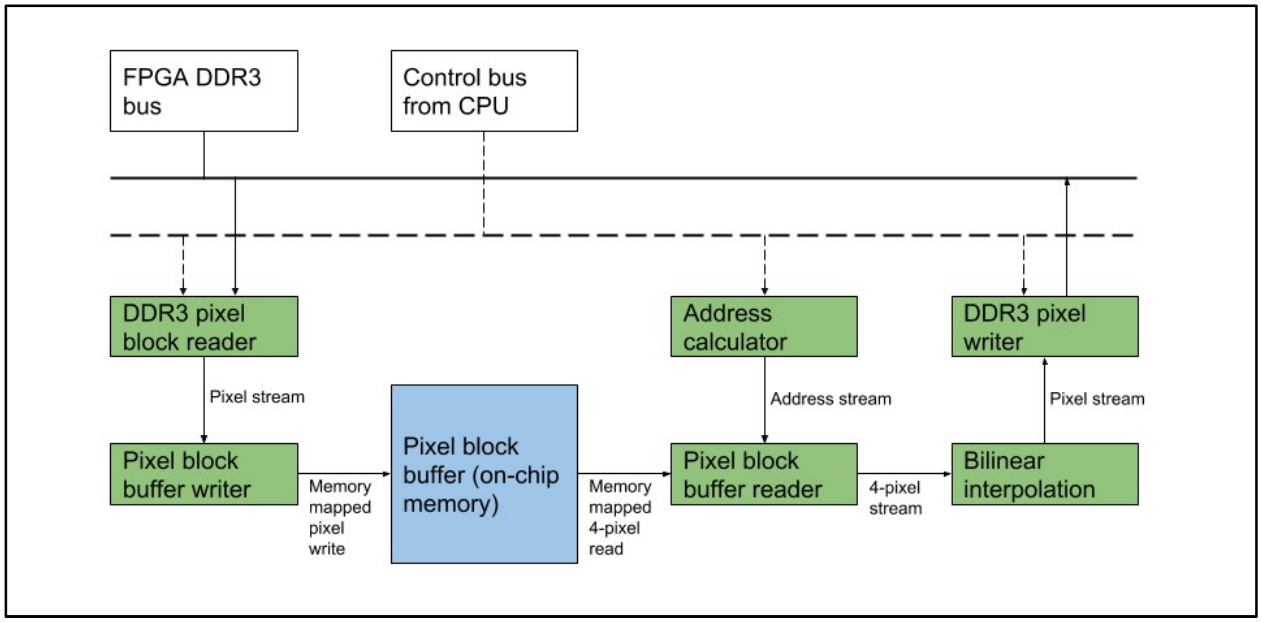

Figure 2.4: Data and control architecture of the lens correction image remapping system.

The address calculator is controlled through the Avalon bus connected to the ARM CPU. The software running on the CPU calculates the coordinates of the source image pixels for each of the four corners of a given block of pixels in the destination image. For our experiments the size of the source image block has been 64 × 64 pixels ($H_b \times W_b$) and the destination block has been 32 × 32 pixels. These sizes can be modified before FPGA compilation to optimize for more extreme distortion or to optimize memory bandwidth. The address calculator receives the four corner coordinates and performs bilinear interpolation to calculate the integer and fractional source coordinates and integer destination coordinates of each pixel in the destination block. These are all streamed to the buffer reader, which uses the integer source coordinates to read the 4 neighboring pixels and stream these, along with the fractional source coordinates and integer destination coordinates, to the interpolation module. The interpolation module receives these 4 pixels and uses the fractional source coordinates to multiply and add the pixels to calculate a

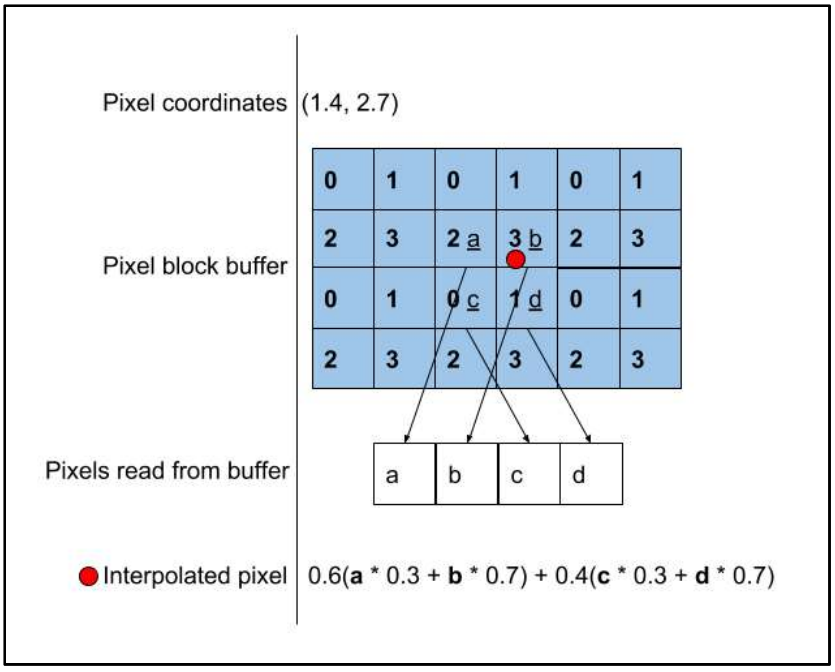

Figure 2.5: Graphical representation of the pixel buffer storage pattern for bilinear interpolation.

resulting interpolated pixel. This interpolated pixel is sent with the integer destination coordinates to the DDR3 pixel writer, which holds the interpolated pixels in a buffer until it receives a full block-width line of pixels and then writes them to the destination image buffer in DDR3.

Where $H_b \times W_b$ are the dimensions of the buffered block from the source image, $H_2 \times W_2$ are the dimensions of the block written to the destination image, and R_{dest} is the number of pixels in the destination image, the total time to remap with this algorithm is expressed as:

$$t_{remap} = \frac{R_{dest}}{H_2 * W_2} ((t_{rand} * H_2 + t_{seq} * H_2 * W_2) + (t_{rand} * H_b + t_{seq} * H_b * W_b))$$

With the same memory parameters previously characterized from the DE10-Standard (19ns t_{rand}, 1.5ns t_{seq}), a source block size of 64 × 64 pixels and a destination block size of 32 × 32 pixels, and a destination image size of 8MP, this results in a remapping time of 74ms - a 58% reduction compared to the naive method. A source to destination block size ratio of 4:1 as used in this example would almost never be necessary except for the most extreme of distortion cases. With a more reasonable 48 x 48 source block size, the remapping time is reduced to 51ms.

2.2.4 Region of Interest Selection

Raw images from the light field array contain a great deal of redundant information that is mostly unused when rendering SAIs. The pixel-weighting algorithm in the VR rendering pipeline ensures that only the pixels captured by the cameras nearest to the viewing position are used to generate the SAI. Allowing the host PC to select regions of interest in the camera images and not receive unnecessary regions can greatly reduce USB 3.0 bandwidth and processing requirements on the host. This functionality is also necessary if we are to connect multiple cameras to one FPGA module that is connected to the host through USB, as the FX3 USB 3.0 bandwidth is limited to

3.2Gbps [21], which would be 75% utilized by a single 5MP camera sending 24-bit RGB images at 20fps.

In our ROI selection implementation, the output image is divided into a grid of arbitrarily sized pixel blocks that are chosen through bit masks sent by the host PC. The grid and block size is chosen at FPGA compilation time. A DDR3 reader module was modified to allow masking of the image to be read. The DDR3 reader still reads the image line-by-line, but it reads a block memory module containing the bit masks to determine which pixels to send and which to ignore. The bit masks are stored in a dual-port on-chip memory block that is written by the state machine module that communicates with the FX3 development kit.

2.2.5 Block Matching

Block matching was implemented on the FPGA as an experimental function for future hardware where multiple cameras would be connected to a single FPGA. This functionality could supplement the GAN-based depth inference or be used for other 3D sensing tasks. Our

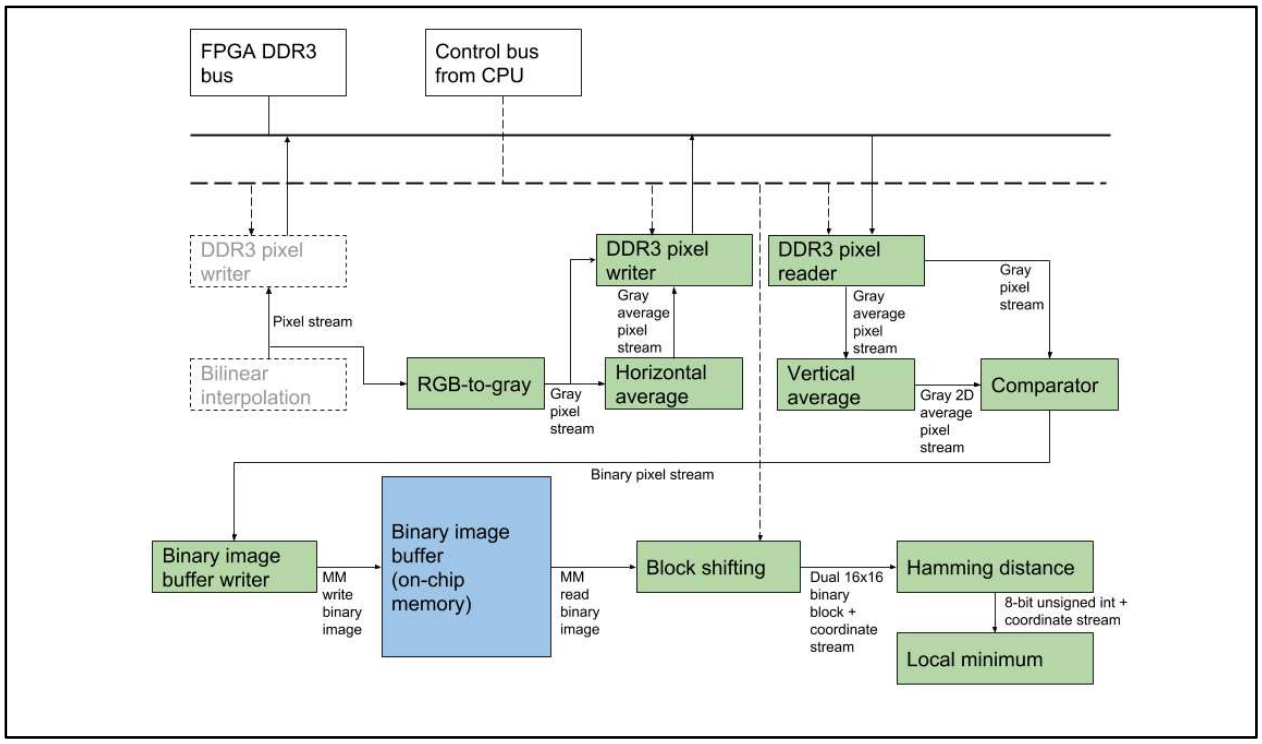

Figure 2.6: Data and control flow diagram of the stereo block matching system.

implementation was designed to be as fast and hardware-efficient as possible, utilizing no multipliers and requiring very little extra DDR3 bandwidth. The block matching module architecture is shown in Figure 2.6.

The pixel stream from the remapping module is tapped and used as the input to the block matching module. The pixels are first converted from 24-bit RGB to 8-bit grayscale and downsampled by 50%, resulting in a 92% reduction in data bandwidth. This stream is then converted to a horizontal average and buffered in the DDR3, with each 16-bit entry in the DDR3 containing the 8-bit grayscale pixel and the 8-bit average grayscale value of the 16 neighboring pixels. This buffer is then read vertically and vertically averaged to acquire a 2D 16 × 16 pixel local average for each grayscale pixel. The grayscale pixels are then compared to their local average and converted to binary, where they are stored in an on-chip memory buffer. This process repeats for the

corresponding image from the other camera in the stereo pair. The two buffers are read by a 2D shift-register module that is controlled by the CPU. The shift-register module outputs two 16 × 16 bit blocks: one representing the search block in one image, and another representing the current block for comparison in the opposite image. The search area used for experimentation was 128 × 32, but this can be modified before FPGA compilation. These blocks are fed to a hamming distance calculator which uses LUTs and adders to get the difference between the binary blocks. Finally, this distance is fed to a local average calculator which finds the minimum hamming distance for a given search block, which can be extrapolated to disparity and therefore depth.

2.3 FPGA Conclusion

Our implementation of real-time image-preprocessing on an FPGA represents an effective proof of concept of a light field image preprocessing system. Each function performed on the FPGA reduces computation time and communications bandwidth to the host PC. This system could eventually be implemented on an FPGA SoM (system-on-module) with higher memory bandwidth

Input resolution	8MP
Input format	10-bit Bayer pattern
Maximum input pixel rate	120MP/s
Image processing functions	Demosaicing, lens correction/remapping, ROI selection, block matching
Output resolution	Variable
Output format	24-bit RGB
Maximum output pixel rate	100MP/s

Table 1: Image Processing FPGA Final Specifications

and more logic resources, allowing multiple cameras to share a single FPGA processing unit. Since each functional IP block is modularized, the system can be easily reconfigured for new hardware platforms and configurations.

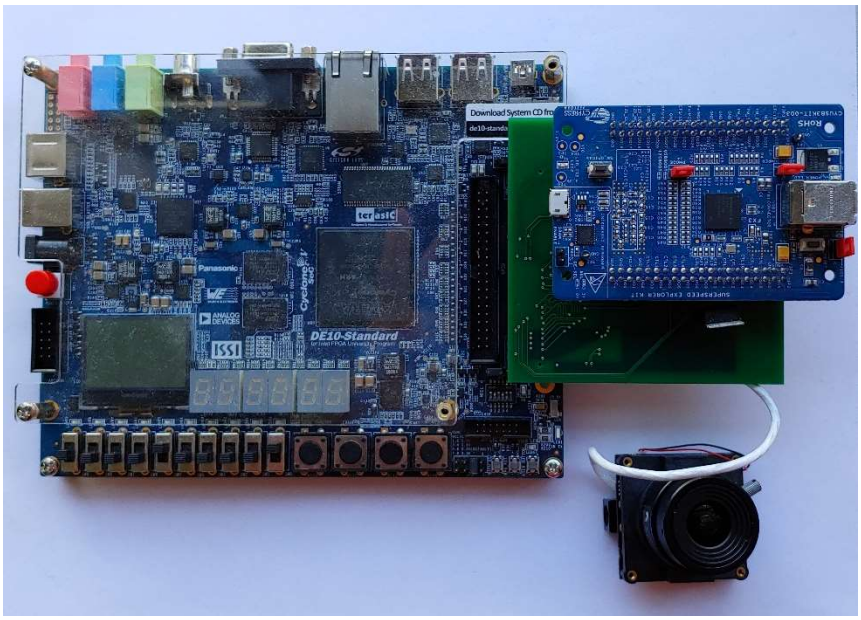

Figure 2.7: The DE10-Standard with the custom PCB attached and connected to the Cypress EZ-USB FX3 and IMX274 image sensor module.

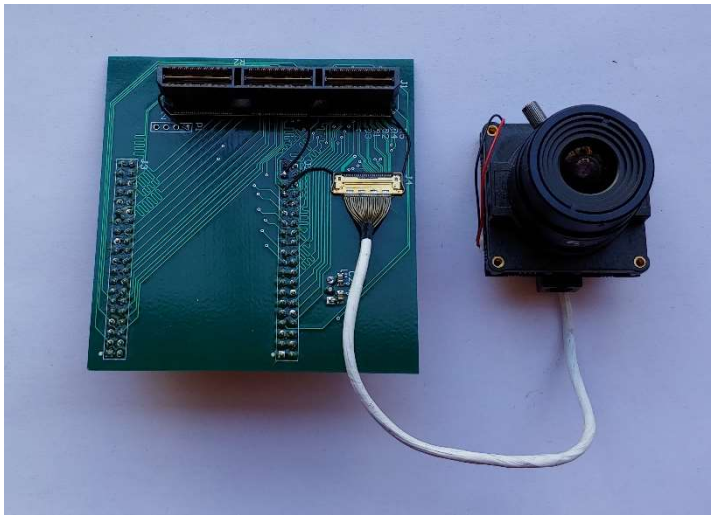

Figure 2.8: The underside of the custom PCB, showing the HSMC and camera I-PEX connectors.

Chapter 3

MACHINE LEARNING

A machine learning approach was selected for our next-generation depth capture system due to its ability to draw from many different cues in the light field image and utilize contextual features to generate a complete depth map. Traditional depth recovery algorithms can be fast and accurate, but struggle with edge cases such as reflective surfaces and thin structures. A machine learning algorithm can be trained to utilize the full structure of a scene and infer depth at points that might be ambiguous if they were analyzed in isolation. Our application requires accuracy sufficient for surgical operations within a cubic space of approximately 150mm per side, located at approximately 600mm from the camera array.

3.1 GAN architecture

The GAN was chosen as a depth map generation network due to its ability to effectively and accurately perform a diverse range of image-to-image translation operations. This ability is a result of the GAN's unique, dual-sided architecture. Training consists of a competition between two neural networks. The first network, the generator, decodes the input image (such as a noisy, inaccurate depth map) and attempts to generate an image as similar to the target as possible. The second network, the discriminator, receives 'real' (target) depth maps or 'fake' depth maps from the generator and attempts to determine which is which. The generator is penalized when the discriminator determines that its depth maps are fake, and the discriminator is penalized when it

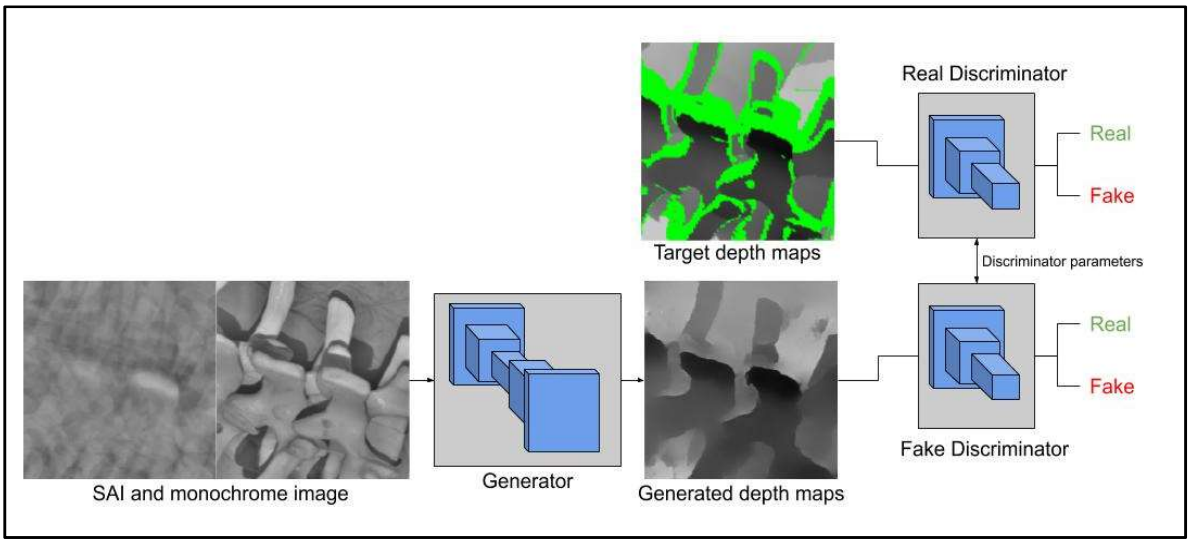

Figure 3.1: The generator and discriminator compete during training. Green areas in the target represent unknown or out-of-range depth. These areas are masked off at the discriminator input. In practice, the discriminator is implemented as two identical modules that share parameters.

misclassifies real or fake depth maps. The generator is also penalized with an L1 loss (absolute difference from target) to enforce accuracy of the output depth maps. The adversarial loss function presents a naturally-learned way to teach the generator how to create realistic depth maps. With most image-generation neural networks, the loss function must be custom-designed, as L1 loss alone has been shown to generate blurry images [22].

3.1.1 Previous experiments

The first version of our GAN was based on the pix2pix architecture [23] which has proven to be successful at many other 2D image translation tasks and required little modification for depth map usage. This network was created using the TensorFlow framework in Python. The 2D GAN was first used to filter depth maps created by algorithms such as block matching and depth from defocus. Both algorithms take advantage of the multiple views of the scene to match features from

different cameras and calculate depth from the disparity of these matches. The raw depth maps from these algorithms were concatenated across channels with a monochrome image from the camera from which the target depth map was captured, shown in Figure 3.1. This combined image was then fed to the GAN, which attempted to produce depth maps with higher resolution and accuracy than the input. This showed promising results, with minimum mean test error reaching 4.15mm out of a 152.4mm range, or 2.7%. However, the two-stage pipeline of depth calculation and GAN refinement created bottlenecks for performance. Both block matching and depth-from-defocus added significant processing overhead to the depth recovery system.

3.1.2 3D GAN

After hitting a performance limit with the 2D GAN, a new architecture was developed which uses the generator neural network to do all implicit depth calculation, rather than refining a pre-calculated depth map. This is implemented through a new 3D generator architecture. In the original pix2pix generator, the 256 × 256 input image is processed through 8 layers of cascaded encoders, each performing 2D convolution with a stride of 2, resulting in the layer size eventually becoming 1 × 1. This is then deconvolved in a symmetrical set of decoders to result in a 256 × 256 output image. Each decoder layer features a fully connected layer to the opposite encoder layer, in order to preserve high frequency features that would otherwise be lost in the dimensionality reduction. This is based on the U-Net architecture [24] shown in 3.2. The 3D GAN replaces the 2D input depth map and 2D encoders with a 3D input and 3D convolutions.

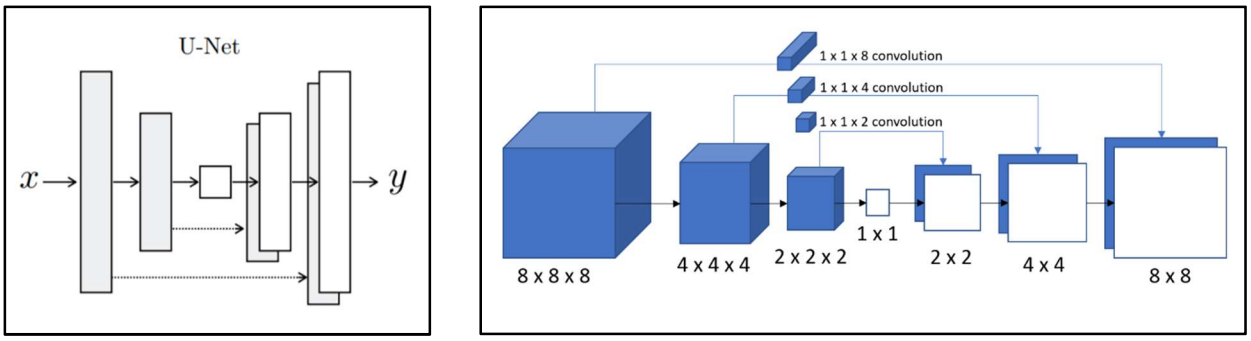

Figure 3.2: Left, U-Net architecture from [24]. Right, truncated adaptation of U-net architecture for 3D encoder filtering.

3.1.3 3D GAN input format

The 3D GAN input is a depth-series of SAIs rendered from the perspective of the camera from which the target depth map is captured. Each layer in an input image refocuses the SAI at a focal plane of incrementally increasing distance, reprojecting the captured images as if the light field was capturing a flat plane at a discrete depth. In odd-indexed SAIs, pixels are the raw grayscale average of rays intersecting with that point in the focal plane, whereas pixels in the even-indexed SAIs are the scaled standard deviation of the rays intersecting with that point. This algorithm is described in Algorithm 1. Objects in the scene near the depth of a given focal plane appear clearly, whereas objects at a dissimilar depth to the focal plane lose coherence and become blurry. The depths of the 2D input images are evenly distributed between a parameterizable minimum and maximum depth, and the target depth is normalized to this range. For training, a depth range of 152.4mm was always used with a minimum depth randomly selected between 584 and 685mm from the projector.

Algorithm 1. GAN Input Generation Pseudocode

```
j = Index of depth map camera
n = Number of cameras
P = n camera projection matrices
K = n camera-to-world coordinate transformation matrices
M = n camera images
d0 = number of depth steps at which to generate SAIs
x0 = x-dimension of target image
y0 = y-dimension of target image
O = output image

for d in [0, d0):
      for x in [0, x0):
            for y in [0, y0):
                  image_points = zeros(n)
                  image_points_valid = zeros(n)

                  for i in [0, n):
                        if i == j:
                              image_points[i] = Mi(x, y)
                              image_points_valid[i] = 1
                        else:
                              // Reproject from 2D x, y to 3D point to 2D
u, v
                              xa, ya, _ = Pi^-1 * [x, y, 1].T
                              p0 = Kj^-1 * Ki * [xa, ya, d].T
                              u, v, _ = Pj * p0
                              if (0 =< u < x0) and (0 =< v < y0):
                                    image_points[i] = Mi(u, v)
                                    image_points_valid[i] = 1

                  // Get the mean and standard deviation of the valid
points
                  mean_pt = sum(image_points) / sum(image_points_valid)
                  dev_sum = 0
                  for i in [0, n):
                        if image_points_valid[i]:
                              dev_sum += (image_points[i] - mean_pt)^2
                  standard_dev = (dev_sum / (sum(image_points_valid) -
1))^0.5

                  if (d % 2):
                        O(x, y) = mean_pt
                  else:
                        O(x, y) = standard_dev
```

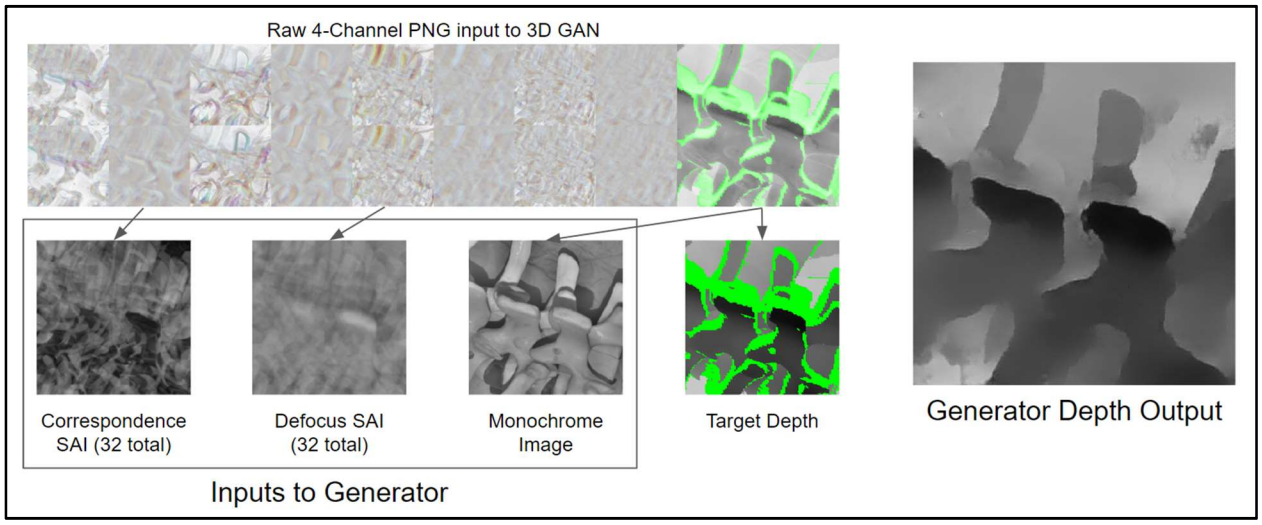

Figure 3.3: Example GAN input, output, and target data. Green areas of target are unknown or out of range.

3.1.4 3D GAN encoder filter

The 3D filters in the encoder layers convolve through the image X-Y as well as Z in the depth direction, finding focused regions and using this to infer output depth. All 3D filters are 4 × 4 in the X-Y direction and 2 deep in the Z direction, with a stride of 2 in X, Y, and Z to create the eventual layer size reduction down to 1 × 1 × 1. Each encoder layer is followed by a feature batch normalization layer and Leaky ReLu activation with slope 0.2. Using 3D convolution slows down the network considerably compared to 2D convolution as the added dimension requires a corresponding increase of multiplications and GPU memory. Training speed using an NVIDIA GP100 is reduced from approximately 18 images/sec with the original pix2pix architecture to 10 images/sec with the 3D encoder architecture.

3.1.5 2D decoder filtering

The decoder layer of the 3D GAN is nearly the same as the decoder of the 2D GAN, with a series of 2D deconvolution layers increasing in size from 1 × 1 to 256 × 256. The filters are sized 4 × 4

with a stride of 2 × 2. The input to each deconvolution is a concatenation of the output from the previous decoder layer and the output from the opposing symmetrical layer of the encoder. However, because the encoder layers are 3D and the decoder layers are 2D, they cannot be directly concatenated as they are in the original architecture. To get around this, each 3D encoder output layer is convolved with filters that are 1 × 1 in the X-Y plane and the same Z-depth as the 3D layer, as a form of learnable dimensionality reduction. Convolving a 3D encoder layer with these filters results in outputs of the same X-Y size as the 3D layer but with a Z-depth of 1. The Z dimension of this layer is removed, and the layer is concatenated with the previous decoder layer to form an input to each deconvolution in the 2D decoder.

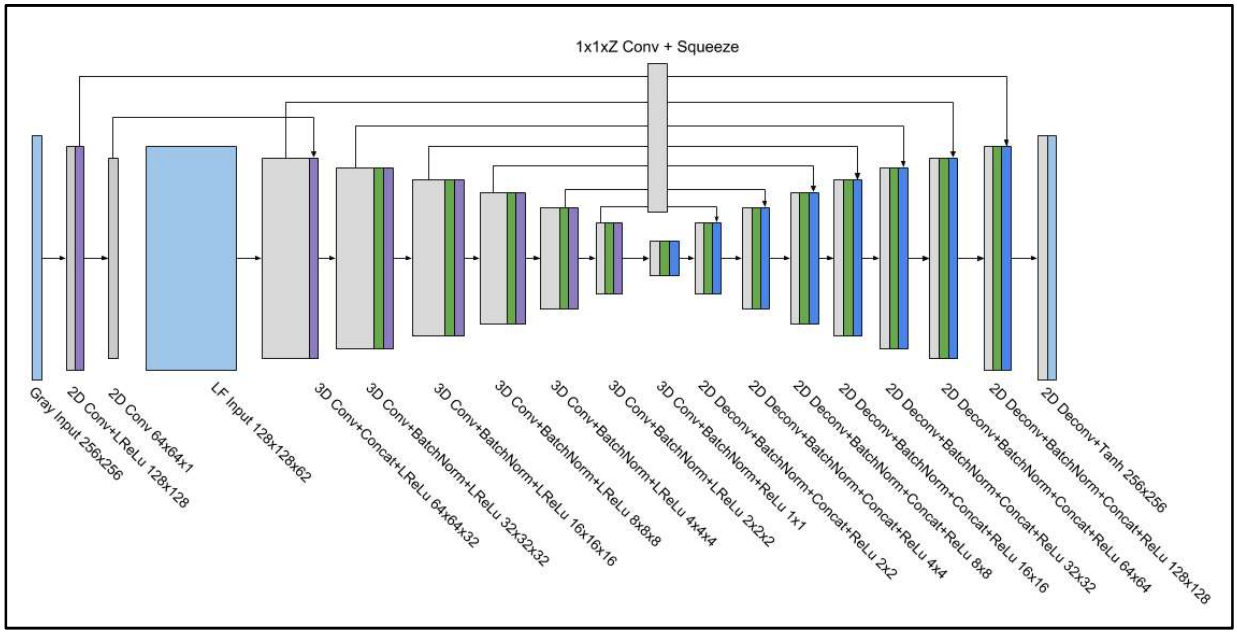

Figure 3.4: Full 3D generator architecture diagram.

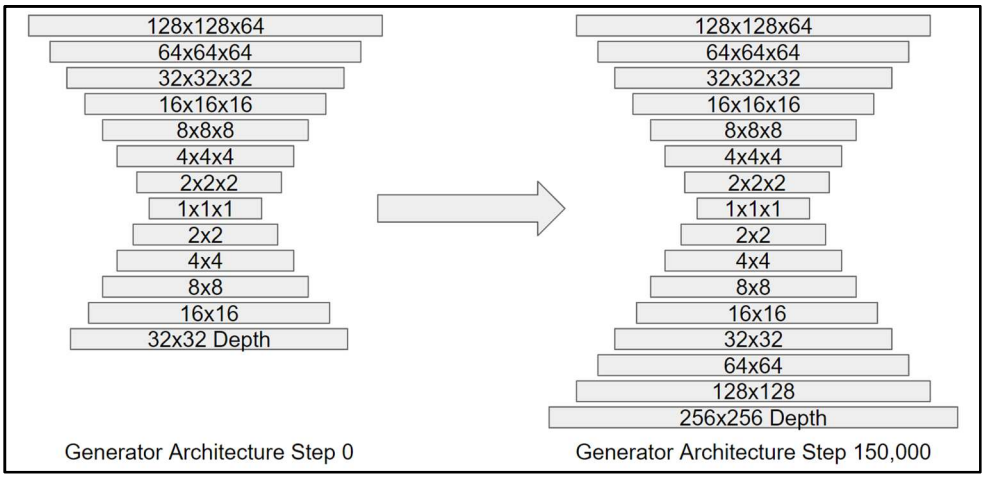

Figure 3.5: Progressive growth training visualization. As training progresses, layers of the generator and discriminator are gradually enabled.

3.1.6 Progressive Training

Hyperparameter searching demonstrated that increasing the weight of the L1 loss term relative to the adversarial loss term resulted in decreases in both test and train L1 error, with little effect on overfitting. In addition, generator outputs showed a great deal of noise and high-frequency artifacts as the discriminator became more effective. This indicated that imbalanced competition between the generator and discriminator may have destabilized training and encouraged overfitting. To

Figure 3.6: Comparison of generator output after 200,000 training iterations without progressive training (left) and with progressive training enabled (right).

counteract this destructive competition, we implemented progressive GAN growth as proposed in Karras, et al. [25]. This training methodology starts with a 32 × 32 pixel generator and discriminator, and progressively enables layers of increasing size as training continues. Layers are gradually introduced to prevent sudden shocks during training. This allows the generator to learn to solve the depth map creation problem in low resolution before being penalized by the discriminator for high frequency errors.

3.2 Target Generation

3.2.1 Mesh Registration

After experimenting with block matching and depth from defocus for depth map creation, capturing more training data, and augmenting training data, it became clear that overfitting was a significant bottleneck for the performance of the GAN. Test error would quickly rise above

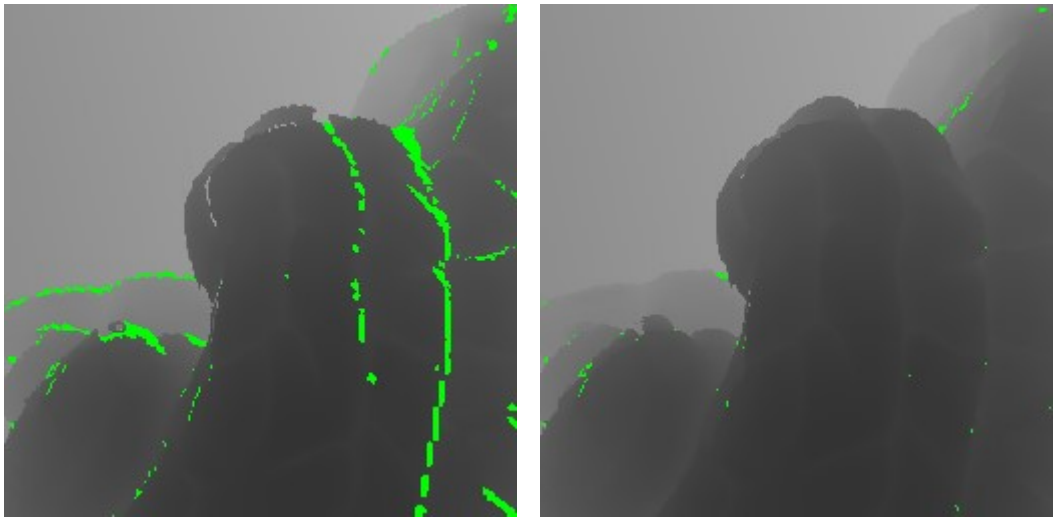

Figure 3.7: Comparison of completeness of a target depth map created from a single scan mesh (left) and a target depth map created from superimposed meshes from multiple scans registered and aligned (right). Green areas represent unknown or out-of-range depth.

training error even with increased amounts of data augmentation, and the generator continued to learn where holes would appear in target data and generate noise in these regions. This indicated that target depth maps needed to be more complete with fewer holes to properly train the GAN and avoid overfitting. The training data had been captured with the camera array on a robotic arm, moving the array to 10 different preset positions around static scenes to increase the number of unique scans while minimizing operator effort. This presented the opportunity to combine meshes from multiple scans of the same scene. Proprio's registration software integrating FPFH [26] and CPD [27] was used to create transformation matrices to combine meshes, and the Blender training data generation software was used to create new depth maps. This resulted in a significant decrease in overfitting and test error.

3.2.2 Synthetic Training Data

The number of unique scans available for training is limited by the time requirements of creating diverse, unique physical scenes for scanning. To get around this bottleneck, we experimented with creating fully computer-rendered scans in Blender. This was implemented through a modification of our depth-map creation software, which imports a mesh into blender, positions the virtual camera at the same perspective as the real camera, and exports the camera depth map as a single precision float image. Multiple free assets from Google's Sketchup Warehouse [28] were added to the scene and a virtual camera was created for every real camera. For every synthetic scan, the virtual 3D assets were randomly repositioned, and a color image was rendered from the perspective of each virtual camera. These images and the depth image from the main camera were processed by the 3D input processing software in the same manner as images from a real scan. A ray-tracing render engine was used in an attempt to achieve photorealistic outputs. These synthetic scans added

Figure 3.8: Comparison of generator output after training with only real scans (center) and generator output after training with both real and synthetic scans (right). The left image is the monochrome image from the camera used for the depth map perspective, showing a segment of a model spine.

training data with extremely high-quality depth maps, resulting in a significant decrease in qualitative noise artifacts at the generator output. These depth maps were complete except for regions which were outside of the normalized depth range. However, the generator L1 loss was much higher on synthetic scans than on real scans, indicating a large discrepancy between the domain of the real images and synthetic images.

Figure 3.9: Example camera image from a synthetic scan.

3.3 Results

Our depth recovery neural network successfully performs depth map generation from light field images without the need for structured light. While we were not able to exceed the accuracy of structured light, our algorithm is much faster and generates complete depth maps. The 3D GAN achieved a marked decrease in error and total depth calculation time compared to depth from defocus and block matching, demonstrated in Table 2. However, the 3D architecture still suffers from some overfitting and noise artifacts in the outputs. These artifacts appear to be mainly caused by the unknown areas in the training data target depth maps. Since there is no depth data available, L1 loss is not applied to these regions and the generator eventually learns where unknown areas will appear and places random data in these areas. This can be prevented by stopping training before the random noise artifacts begin to appear. Increasing the amount of synthetic data used in training also reduces this behavior but increases the output error of the generator on real scans. Most unknown areas are regions where the original target depth map is outside the normalized depth range, but some holes are present on sharp edges and translucent or reflective objects. Thresholding training images by the percentage of unknown target depth pixels helped to reduce generator noise artifacts but also decreased the size of the training set, necessitating a compromise

Algorithm	Depth formation CPU time (ms)	GAN inference time (ms)	Test L1 error (mm)	Test L1 error w/GAN (mm)
Structured Light	800 (GPU)	N/A	0.1	N/A
Block matching	2200	52	5.51	3.96
Depth from defocus	3500	52	6.63	4.88
3D GAN	N/A	65	N/A	2.36

Table 2: Comparison of mean test error of raw depth calculation and GAN-filtered raw depth maps.

between the completeness of the training images and the size of the training set. For most experiments, training images were filtered to have a maximum of 32% unknown pixels in the target depth map, resulting in about 40% of training images being removed from the original training set. Our training set consists of 1,050 individual scans from different camera positions of 105 unique scenes. Through data augmentation, these scans were transformed to about 4,000 unique training images from real scans. These images were randomly mixed with synthetic images for training.

3.4 Machine Learning Conclusion

We hope to continue this work and reach for better accuracy by stacking a second neural network to refine the output of the first, an enhancement that has repeatedly been shown to improve the output accuracy of machine learning algorithms [29]. The output at time of writing is sufficiently accurate to be used for registration to CT scans using the Proprio registration software, an important function of the light field system.

Inference time*	65ms
Output resolution	256 × 256
Precision	32-bit float
Minimum measured depth	584.2mm
Maximum measured depth	838.2mm
Standard depth output range	152.4mm
Mean absolute error in 152.4mm range	2.36mm
RMS error in 152.4mm range	5.19mm

*After input image generation

Table 3. Final specifications of 3D Depth GAN.

Chapter 4

CONCLUSION

This work focuses on the development of two novel processing tools for light field data - FPGA image pre-processing and GAN depth acquisition. The FPGA image processing platform reduces hardware complexity and removes processing load from the host PC, allowing faster rendering and freeing up processing power for other tasks such as the GAN. The GAN model also reduces hardware complexity while capturing more complete depth maps in less time. In combination, these systems can increase the quality and resolution of images generated from the LFS in real time, while decreasing latency, cost, and complexity.

BIBLIOGRAPHY

[1] Tahmaseb, Ali, et al. "The accuracy of static computer-aided implant surgery: A systematic review and meta-analysis." *Clinical oral implants research* 29 (2018): 416-435.

[2] Levoy, Marc. "Light fields and computational imaging." *Computer* 8 (2006): 46-55.

[3] Lee, Jiwon, Mingyu Kim, and Jinmo Kim. "A study on immersion and VR sickness in walking interaction for immersive virtual reality applications." *Symmetry* 9.5 (2017): 78.

[4] Gu, Qing, Kyriakos Herakleous, and Charalambos Poullis. "3dunderworld-sls: An open-source structured-light scanning system for rapid geometry acquisition." *arXiv preprint arXiv:1406.6595* (2014).

[5] "Intel® RealSenseTM D400 Series (DS5) Product Family Datasheet." Intel Corporation, 1 Jan. 2018.

[6] Khoshelham, Kourosh, and Sander Oude Elberink. "Accuracy and resolution of Kinect depth data for indoor mapping applications." *Sensors (Basel, Switzerland)* vol. 12,2 (2012): 1437-54. doi:10.3390/s120201437

[7] Scrofano, Ronald, Seonil Choi, and Viktor K. Prasanna. "Energy efficiency of FPGAs and programmable processors for matrix multiplication." *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.*. IEEE, 2002.

[8] "AN 754: MIPI D-PHY Solution with Passive Resistor Networks in Intel® Low-Cost FPGAs." Intel Corporation, 3 Apr. 2019.

[9] Shah, Dave. "Open Source 4k CSI-2 Rx Core for Xilinx FPGAs." *GitHub*, 1.0, 14 Nov. 2018, github.com/daveshah1/CSI2Rx.

[10] Li, Xin, Bahadir Gunturk, and Lei Zhang. "Image demosaicing: A systematic survey." *Visual Communications and Image Processing 2008*. Vol. 6822. International Society for Optics and Photonics, 2008.

[11] Rani, K. Sudha, and W. Jino Hans. "FPGA implementation of bilinear interpolation algorithm for CFA demosaicing." *2013 International Conference on Communication and Signal Processing*. IEEE, 2013.

[12] Bailey, Donald, Sharmil Randhawa, and Jim S. Jimmy Li. "Advanced Bayer demosaicing on FPGAs." *2015 International Conference on Field Programmable Technology (FPT)*. IEEE, 2015.

[13] Greisen, Pierre, et al. "An FPGA-based processing pipeline for high-definition stereo video." *EURASIP Journal on Image and Video Processing* 2011.1 (2011): 18.

[14] Häublein, Konrad, Marc Reichenbach, and Dietmar Fey. "Fast and generic hardware architecture for stereo block matching applications on embedded systems." *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, 2014.

[15] Mazumdar, Amrita, et al. "A hardware-friendly bilateral solver for real-time virtual reality video." *Proceedings of High Performance Graphics*. ACM, 2017.

[16] Ryan Fanello, Sean, et al. "Hyperdepth: Learning depth from structured light without matching." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016.

[17] Gwn Lore, Kin, et al. "Generative adversarial networks for depth map estimation from RGB video." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2018.

[18] Tao, Michael W., et al. "Depth from combining defocus and correspondence using light-field cameras." *Proceedings of the IEEE International Conference on Computer Vision*. 2013.

[19] Yan, Tao, et al. "Depth Estimation From a Light Field Image Pair With a Generative Model." *IEEE Access* 7 (2019): 12768-12778.

[20] Malvar, Henrique S., Li-wei He, and Ross Cutler. "High-quality linear interpolation for demosaicing of Bayer-patterned color images." *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 3. IEEE, 2004.

[21] "EZ-USB® FX3: SuperSpeed USB Controller." Cypress Semiconductor Corporation, 17 Dec. 2018.

[22] Kristiadi, Agustinus. "Why Does L2 Reconstruction Loss Yield Blurry Images?" *Agustinus Kristiadi's Blog*, 9 Feb. 2017, wiseodd.github.io/techblog/2017/02/09/why-l2-blurry/.

[23] Isola, Phillip, et al. "Image-to-image translation with conditional adversarial networks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

[24] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." *International Conference on Medical image computing and computer-assisted intervention*. Springer, Cham, 2015.

[25] Karras, Tero, et al. "Progressive growing of gans for improved quality, stability, and variation." *arXiv preprint arXiv:1710.10196* (2017).

[26] Rusu, Radu Bogdan, Nico Blodow, and Michael Beetz. "Fast point feature histograms (FPFH) for 3D registration." *2009 IEEE International Conference on Robotics and Automation*. IEEE, 2009.

[27] Myronenko, Andriy, and Xubo Song. "Point set registration: Coherent point drift." *IEEE transactions on pattern analysis and machine intelligence* 32.12 (2010): 2262-2275.

[28] "Google 3D Warehouse." *3D Warehouse*, Google, Inc., 2019, 3dwarehouse.sketchup.com/?hl=en.

[29] Breiman, Leo. "Stacked regressions." *Machine learning* 24.1 (1996): 49-64.