CHALMERS

Implementing stereoscopic video processing on FPGA

Master's thesis in Embedded Electronic System Design.

Ástvaldur Hjartarson, Klas Nordmark
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, June 2015

Implementing stereoscopic video processing on FPGA
Ástvaldur Hjartarson and Klas Nordmark

Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A frame processed by the system described in this report, taken after (from left to right) remapping, Gaussian blur and Sobel filtering.

Typeset in LaTeX
Department of Computer Science and Engineering
Gothenburg, Sweden June 2014

Abstract

The purpose of this thesis is to investigate the viability of using FPGA acceleration in the processing of a stereoscopic video feed. This is done by comparing speed for a given processing resolution with a software implementation, as well as investigating power and area usage. The processing performed include greyscaling, remapping, resizing, Gaussian blur and Sobel filtering. Methods for disparity map calculations are also investigated. A system capable of processing video at 60 stereo frame pairs per second was developed.

Keywords: computer vision, FPGA, video, image processing, disparity, stereoscopy.

Acknowledgements

We want to thank Lars Svensson at the Chalmers Department of Computer Science and Engineering for his support, as well as Johan Wiebe for his invaluable knowledge of the underlying mathematics of this work. We are also grateful towards Travis Oliphant for the original creation of NumPy, which has helped us immensely as a free tool for mathematical programming.

Ástvaldur Hjartarson and Klas Nordmark, Gothenburg, June 2015

Contents

List of Figures

List of Tables

Nomenclature

BRAM Block Random Access Memory.

B One byte.

DDR3 Dual Data Rate 3, a kind of dynamic random access memory.

Disparity map Matrix containing information on the coordinate difference between points in a stereo image pair.

Epipolar plane The plane formed by a point in 3D space and the optical centers of two cameras.

FF Flipflop, one-bit storage element.

FPGA Field Programmable Gate Array, a kind of programmable logic device.

Frame An image in a video feed.

Gaussian blur Smoothing filter based on the Gaussian distribution.

Horopter is the 3D space that the stereo camera explores.

Kernel Matrix used in convolution with an image.

LUT Look-up table.

Pixel Smallest element of a frame, provides color information for a position.

RGB Red Green Blue, additive color model based on these three colors.

SAD Sum of Absolute Differences, a method for quantifying similarity.

Sobel filter A kind of edge-enhancing filter.

Stereo image pair Two images of the same scene taken from slightly different positions.

1
Introduction

The purpose of this thesis is to investigate the viability of using FPGA acceleration in the processing of a stereoscopic video feed. This is done by comparing speed for a given processing resolution with a software implementation, as well as investigating power and area usage. Of particular interest is investigating methods for disparity map calculations.

Video processing is typically a very computationally heavy task, with each image frame providing a large amount of data. For example, a direct implementation of convolution between an image of 1024 by 768 pixels and a 7 by 7 elements large matrix needs 49 multiplications and 48 additions for each output pixel. For a frame rate of 20, this would require 1526 million integer operations per second. As we cascade several tasks of comparable complexity, software on general embedded processors may become unsuitable for real-time applications [1].

Many image processing algorithms, however, are inherently parallel. In the example above, no output pixel is dependent on any other output pixel. In dedicated hardware, all 49 multiplications for an output pixel could be performed in a single clock cycle. If very high speeds were required, multiple output pixels could be calculated concurrently. In a system with cascaded image processing, the different processing stages could also be performed concurrently in a pipeline. As custom silicon solutions are not reconfigurable and have prohibitively large non-recurring engineering costs, FPGA becomes an attractive choice of platform.

The thesis is structured as follows. An overview of the method that the problem was approached with is given in Chapter 2, Methodology. Chapter 3, Theory, provides a mathematical description of all processing stages that are to be implemented. Chapter 4, Implementation, provides information on the general architecture of the system that has been developed, the interfaces that are used and the implementation of the algorithms described in Chapter 3. Results such as calculation speed and area usage are provided in Chapter 5. Chapter 6 provides discussion on the results and problems that have been encountered, and Chapter 7 contains the conclusions that have been reached.

1.1 Background

Research in real-time video processing for extracting useful data is utilized in many fields, such as active-safety systems and surveillance [2][3]. The computational complexity of the algorithms involved lead to specialized hardware being used, such as digital signal processors or FPGAs [3].

Using an FPGA to accelerate video processing in general is a well-documented area [3]. Problems similar to what will be described below can be found in literature [4][5], and our scientific motivation is to investigate if we can make any improvements in terms of speed compared to existing solutions.

1.2 Aims

During the course of the thesis project, a video preprocessing system will be developed. The intended application is for use with stereoscopic cameras in the context of pattern recognition, particularly the detection of humans in the video feed. It is expected that the depth information will increase accuracy of pattern recognition compared to a simple 2D image.

Specifically, the project aims to implement the following processing stages in an FPGA:

- Conversion from RGB to greyscale, in order to reduce the amount of data.
- Remapping, in order to to compensate for distortion introduced by the optics and aligning rows for disparity calculation
- Resizing, in order to reduce the amount of data.
- Smoothing with a Gaussian blur, in order to reduce the impact of high-frequency noise
- Sobel filtering, in order to enhance edges

For the entire system, and where applicable for each block, the following questions can then be investigated:

- How many clock cycles are needed to perform the processing?
- How much area does the implementation use?
- What is the minimum memory bandwidth needed?
- How much power is dissipated?
- What is the maximum stereo frame rate possible?
- What is a suitable amount or parallelism, considering platform constraints and the throughput of other processing stages?

All of this put together let us answer the final question - is FPGA a suitable platform for the problem at hand? The goal is to process a minimum of 20 stereo frame pairs every second. Higher frame rates would be preferable. In order to use this system, I/O and memory interfaces will have to be developed as well.

The report is also intended to provide a small study of different methods for disparity map calculations, in order to make a recommendation on which algorithm to use in further development. The disparity map is calculated from a stereo image pair and gives depth information.

1.3 Limitations

The method used in this study is to implement the processing on a particular FPGA development board - it is deemed too time-consuming and of limited academic interest to design a custom PCB. Accordingly, the project has strict limits defined by the components available on the board. Field tests with an actual camera are outside the scope of the project.

1.4 Material

An FPGA platform and associated software was necessary in order to execute the project. This section describes the tools used and points out important features.

1.4.1 Hardware

The development board used in the project was an Xilinx Artix-7 FPGA AC701 Evaluation Kit [6]. Components of particular interest on the board are

- Artix-7 XC7A200T FPGA from Xilinx featuring 215360 logic cells, 740 DSP slices and 13140 kbit of on-chip RAM [7].
- DDR3 memory from Micron Technology, 1 GB large with 64-bit wide bus and a bandwidth of 12.8 GB/s [8].
- I/O including 1000 Mbit/s ethernet connection and HDMI output.
- A Quad SPI Flash from Micron containing 32MB (256Mbit).

An Bumblebee stereo camera was used to film the video used to test the pipeline [9].

1.4.2 Software

The Xilinx Vivado Design Suite was used as the development environment throughout the project. Apart from supporting implementing and programming designs for the device that was used [10], it offers helpful features such as simulations [11], creating and reading integrated logic analyzers (ILA) for debugging [12], and a catalog of IP cores from Xilinx [13]. Xilinx SDK was used to develop software for a Microblaze core used for verification purposes.

2
Methodology

This chapter describes the initially planned methods for developing the system and finding answers to the questions described in Chapter 1. First the initial literature study is described. Following this, the approach to the design process is presented. Finally, the planned evaluation methods are described.

2.1 Literature study

The project began with a literature study in order to understand the problem at hand. Relevant sources were found primarily through the search tools of the Chalmers library. Both established literature on digital image processing and recent research papers, particularly on disparity calculations, were searched for. Manuals and data sheets provided online by Xilinx were also very important sources of information. A full list of sources used in the writing of this report can be found in the bibliography.

2.2 Design methodology

Platform choice was made before the project formally began, based on rough calculations on the required memory bandwidth and FPGA size (particularly the available on-chip RAM and number of multipliers).

While it was expected that many design choices would have to be determined after the literature study was done, a rough idea of how to approach the problem was determined at the start of the project. The system itself would consist of a series of processing stages with some kind of synchronization method. Use of a high-speed external memory would be necessary - the DDR3 on the board chosen. It was also known that the greyscaling and downsizing of the image would decrease the amount of data to such a degree that much of the intermediate data could be stored on-chip on the current platform, improving speed and avoiding arbitration issues for DDR3 access.

Another important requirement for the project that was known from the beginning was the need of a way to write and read images to the system. As the development board does not have an image sensor, we would need to upload images from a computer. The initial plan was to communicate over UART, with storage being done on the on-board Flash memory.

Our planned debugging method was mainly to use the Vivado simulator. We also planned to use integrated logic analyzers when simulated behavior and on-chip behavior did not match.

2.3 Evaluation

Evaluation was to be performed by determining throughput, latency, area and power consumption with help by the Vivado IDE, both for individual processing stages and the system as a whole. While the primary goal was to meet the minimum frame rate of 20, investigating these quantities for different amounts of parallelization was considered interesting as well. The input to the system in simulation was an recording from an Bumblebee stereo camera [9]. The input would then give a reasonable activity file that could be used to determine the power consumption.

Answering the main question of the thesis - if an FPGA is a suitable platform for the application or not - would be done by comparing the cost of the device with the cost of a CPU or GPU capable of meeting the same constraints on throughput. Software implementations were available for this comparison. Reliability and power consumption would be secondary aspects to compare.

Part of the evaluation would also be to theoretically reason about a few different algorithms for disparity map calculations and determine which one of these would be most suitable for the application and the target hardware.

3

Technical background and theory

This chapter provides technical background and mathematical background. First some important terms related to digital video in general and stereo video in particular are defined, and then the theory behind all processing stages is described. Finally, the problem of disparity map estimation is described. Two different algorithms are presented.

3.1 Digital video

Video consists of a series of still images — called frames — displayed in fast enough succession to give the impression of fluid movement. In the digital domain, the frames are matrices of pixels. The pixel is the smallest area for which color and intensity can be defined in the image [14]. Keeping with the fact that the image is mathematically a matrix, terms such as row and column will be used throughout the thesis.

There are many different methods to represent color, known as color spaces. The one most commonly used in displays is the RGB color model. In this model, the color of a pixel is defined as the sum of a red, a green, and a blue component, each represented with a certain number of bits, commonly eight [14]. Input data has been assumed to be represented in this way during the project. Internally the system mostly uses a greyscale representation, meaning that each pixel is only represented by an eight bit intensity value.

Convolution is an important operation in digital image processing. Two-dimensional discrete convolution is a straightforward extension of the one-dimensional case. A function is "slided" over the image, with the output pixel at the corresponding position being a linear combination of the all pixels below the function scaled by the value of the function at that position. In any practical case, the function with which the image is convolved would be windowed and thus a matrix of finite size. In image processing, this matrix is often called a kernel [1]. The convolution operation is shown in Equation 3.1, where I is an image, k is a kernel, and K is the domain of k [15].

$$(I * k)(r, c) = \sum_{(i,j) \in K} I(r - i, c - j)k(i, j) \tag{3.1}$$

3.1.1 Stereo video

The system described in this thesis is intended for usage with a stereo camera. A stereo image pair consists of two images of the same scene, taken from slightly different positions at the same time. The differences in the images can be used to calculate distances [15]. A few terms will be explained here in order to facilitate the understanding of later sections, particularly on disparity map calculations.

The problem of extracting depth information from a stereo video feed is greatly simplified if each row in the left image resides on the same epipolar plane as the corresponding row on the right image. An epipolar plane is a triangle with its points at the optical centers O_L and O_R of the cameras and some point P. The lines containing the intersection between the epipolar plane and the image planes are known as epipolar lines [16]. The concept is illustrated in Figure 3.1.

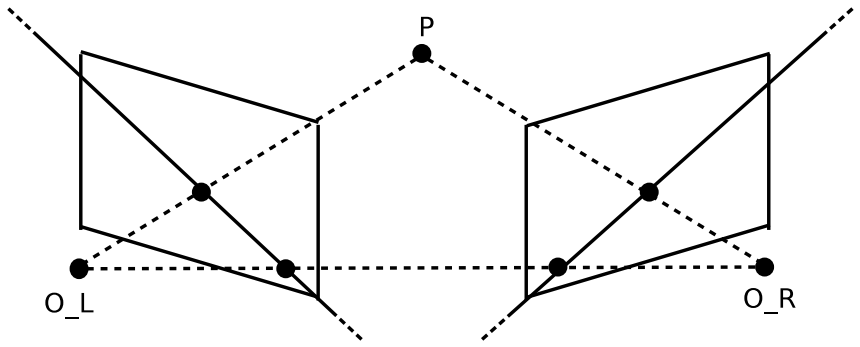


Figure 3.1: Illustration of epipolar geometry.

Generally, a point in space is projected to different coordinates on the left and right image planes. This difference is known as disparity and is useful for determining distances in the images. If the epipolar constraint described above is met, the disparity between the images only needs to be calculated along one dimension [16]. Meeting this condition is one of the purposes of the remapping stage described later.

3.2 Conversion from RGB to greyscale

Color data is not considered to be of importance in the system, only intensity. Therefore, all images are converted to greyscale in order to decrease the amount of data processed by later stages. The conversion from RGB to greyscale is made by adding together the different color values with different weights according to Equation 3.2:

$$D(i,j) = 0.2989R(i,j) + 0.5870G(i,j) + 0.1140B(i,j) \tag{3.2}$$

Here R, G, and B are matrices containing the red, green and blue intensity values for the input frame and D is the matrix representing the output frame.

The greyscaling is done for the left and right images independently. The weights chosen are based on the ITU-R Recommendation BT.601 for digital encoding of video [17].

3.3 Remapping

In stereo vision systems where frames are compared to each other, especially in disparity calculations, it is important that the frames do not have any unwanted difference which could distort the comparison. The difference can be due to lens distortion and also because the camera sensors are not necessarily perfectly aligned. As mentioned in Section 3.1.1, calculations are made simpler if the epipolar constraint is met. The cameras are spaced a few cm apart and may not be exactly parallel to each other. Therefore the frames go through a remapping stage where the frame is rectified and undistorted. The distortion that has to be compensated for is inherent in the stereocamera lens.

There are two main sources of distortion in the Bumblebee camera lens: radial distortion and tangential distortion. Radial distortion is due to the shape of the magnification lens. The magnification in the middle of the image stretches the observed area, but as we go near the edges of the image, observed area is compressed. Tangential distortion is an effect that appears when the lens is not exactly parallel to the imaging plane of the CMOS sensors. This makes the image a little skewed or tilted. To compensate for the distortion in the lens, a calibration matrix is computed offline before the system is put online. The calibration is done as described by Zhang [18].

We rectify the image so that each row in each stereoscopic frame pair has matched epipolar lines [19], which means that a matching pixel pair that points to the same geometric area or object should be found on the same row in each frame. This requirement greatly simplifies the search for matching pixels. Before rectification, the image sensors are not coplanar, since image sensors might have some manufacturing issues. Then, rectification is performed and it uses geometric transforms to rotate the images so they are coplanar and the rows in each frame are aligned according to the epipolar constraint [20].

The undistortion of the lens and the rectification of the frames are combined into a transformation matrix. The transformation matrix can be split into two parts: forward mapping and backward mapping. A demonstration of forward mapping can be seen in Figure 3.2. The input image is shown, with parts of the image that will be removed in the remapping red-tinged. The forward mapping is then essentially the gray part of the image that is left. The next step is to backward map using these pixels to produce the output frame. The backward mapping is to map an output pixel to a pixel in the forward mapped region. The position in the forward mapped region that is mapped to does not necessarily coincide with a pixel. Subpixel information is used, making interpolation necessary. For example, in Figure 3.3, the output pixel at position (1, 2) is supposed to take its value from position (115.6, 49.3) in the original image, where $x = 0.6$ and $y = 0.3$ are fractional components. The intensity of the output pixel is therefore calculated from the neighboring pixels through bilinear interpolation:

$$U = a(1-x)(1-y) + bx(1-y) + c(1-x)y + dxy \tag{3.3}$$

U is then placed in position (1,2) in the output frame. The bilinear interpolation of Figure 3.2 can be seen in Figure 3.4.

Figure 3.2: Demonstration of forward mapping.

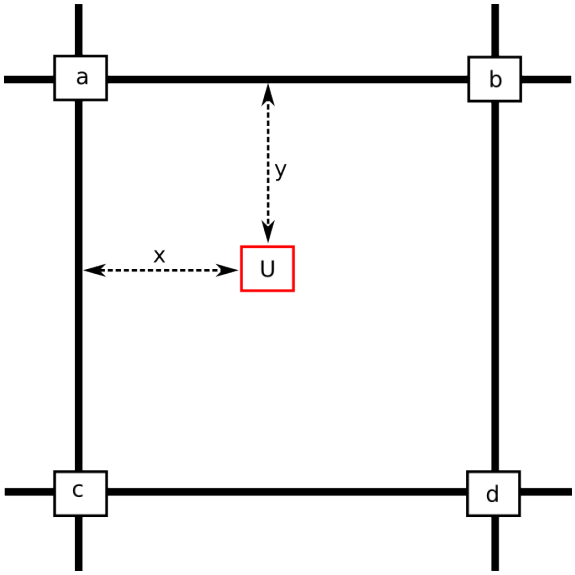


Figure 3.3: A demonstration of the bilinear interpolation.

Figure 3.4: Software implementation of the output from the remapping stage.

3.4 Resizing

In order to decrease the amount of calculations needed in the later parts of the system, the frames are downscaled. While a lower resolution camera feed could be used to begin with, having a higher resolution input has advantages - the higher-resolution image could be useful in finer feature detection, making it possible to implement more functionality later on. The resizing of the images uses a weighted average to calculate the values for the pixels in the smaller image. This is done first either along the rows, or the columns of the image matrix, and then along the other dimension [14]. For each row i_b in an intermediate, row-downsized output matrix B, the value of each pixel can be calculated according to Equation 3.4.

$$B(i_b, j) = \sum_{k=-u}^{u} w(k)A(i_a + k, j) \tag{3.4}$$

Here A is the input image frame and i_a is the middle of the rows being averaged for i_b. u is equal to half the length of the nonzero part of the weight function $w(k)$ rounded down. The same procedure can be done along the columns of the intermediate matrix to complete the downsizing.

The weight function used is triangular in shape. The exact weights are different for each output pixel. An output pixel corresponds to an area in the original image whose center does not necessarily coincide exactly with a certain input pixel, and different weights are used to take this sub-pixel information into account. The idea is illustrated in Figure 3.5. The weights are calculated by assuming a triangle window centered around the corresponding position of the output pixel in the original image with sub-pixel precision. The triangle function has value 1 at the center and then decreases linearly to zero at the edges. The weights are then calculated by finding the value of the triangle function at the original pixels within the window, and then normalizing the sum to 1.

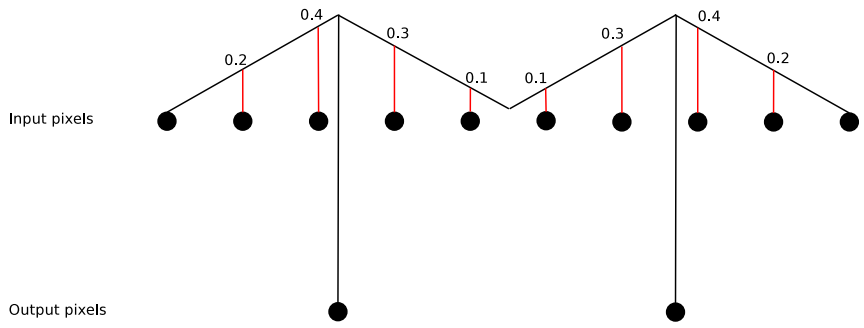


Figure 3.5: Illustration of weight calculation.

3.5 Filters

After resizing, the images are filtered in order to smooth out high frequency noise and emphasize edges. The smoothing is achieved with a Gaussian blur, and the edge enhancement is done with a Sobel filter.

3.5.1 Gaussian blur

This filter consists of convolving the image with a windowed two-dimensional Gaussian function, shown in Equation 3.5. The filter prevents outliers and other noise that may have been introduced in the image capture and processing steps thereafter [21].

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{x^2+y^2}{2\sigma^2}} \tag{3.5}$$

This constitutes a low-pass filter, smoothing the image. The window used in this work has a size of seven by seven pixels. The image frame is zero-padded to enable calculations for the pixels along the edges of the frame. If we denote the input matrix A and the output matrix B, the operation can be expressed as in Equation 3.6.

$$B(i,j) = \sum_{m=1}^{7} \sum_{n=1}^{7} A(i+m-1, j+n-1) G(m,n) \tag{3.6}$$

3.5.2 Sobel filter

Edges in an image are characterized by fast changes in intensity. Accordingly, a filter that emphasizes edges should leave high-frequency components as they are while attenuating low frequencies. One of the simpler ways to do this in digital image processing is with a Sobel filter. The Sobel filter is a discrete-time approximation of the gradient of the continuous image that the frame is assumed to be sampled from. The spatial derivative of an input matrix A is approximated in the horizontal and the vertical direction through convolution with two different kernels, as shown in Equation 3.7 and Equation 3.8, respectively [22].

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A \tag{3.7}$$

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A \tag{3.8}$$

With this convolution, each output pixel in G_y and G_x is a weighted sum of the difference between its immediate adjacent pixels along the respective dimension and the differences between the corresponding pixel pairs immediately adjacent to this pair. The middle pair is given weight 2 while the two other pairs are given weight 1.

In this application, only the magnitude is important, not the direction of the gradient. The magnitude is calculated as shown in Equation 3.9.

$$G(x,y) = \sqrt{G_y^2(x,y) + G_x^2(x,y)} \tag{3.9}$$

3.6 Disparity map

The disparity map is a matrix of all the disparities that can be calculated between the two cameras. To find the pixels that correspond to the same point in space is a computationally heavy operation, and we will go into the details later in the section. But if the we know which pixels match then distance can be calculated. Figure 3.6 demonstrates a stereo camera setup, where distance z to point h is related to the disparity between frames [20]. Disparity is $d = x_L - x_R$, x_L and x_R are the position of the best matching pixels in each frame. The left image plane and the right image plane are assumed to be coplanar, which should be true after the remapping stage. Distance f_L and f_R is the length from the focal point(O_L and O_R) of the lenses to the image plane of each camera, $f = f_L = f_R$ is true if the planes are coplanar. Baseline b stands for the length between the two camera centers. The pixel size (p) of each individual pixel in the camera also affects the length, since the pixel gives discrete intensity values at a fixed area and therefore affects the continuity of the baseline. The focal length has also to be tuned to the pixel size, since too much light going through individual pixels sensors will distort the image. B and C in Figure 3.6 are then finally assumed to be of equal length and with these assumptions we can derive the length from the baseline b to the point h.

$$\mathrm{z} = \frac{bf}{dp} \tag{3.10}$$

As can been seen in Equation 3.10 there is an inverse relationship between the disparity and distance. Therefore a small disparity represents a great distance and in contrast a large disparity represents a small distance, this relation can be seen in Figure 3.7. In Figure 3.7 we plot the relationship between disparity and distance with parameters from a Bumblebee stereo camera [9].

When deciding which algorithm is feasible for detecting objects such as humans, design accuracy will be an important factor. In Figure 3.7, five black lines are plotted. They represent the distance that can be achieved with different disparity accuracy, if we want to be able to have at least 10 cm depth resolution. Furthermore the lines represent the minimum allowed disparity, that is if resolution of 10 cm is needed. With better accuracy, the running time of the algorithms is usually slowed down. However, the camera can be set up so that small inaccuracy will not affect the range of intrest. On the other end is the maximum disparity and it can limit how close to the camera we can see. The higher the maximum disparity is the slower the algorithms become, since each pixel has to compare with more pixels in the other frame. After the rectification step in remapping, the rows in each camera frame should be epipolar lines and therefore the matching location in the other image should be along the same row. Hence the task of finding a matched pixel is greatly simplified.

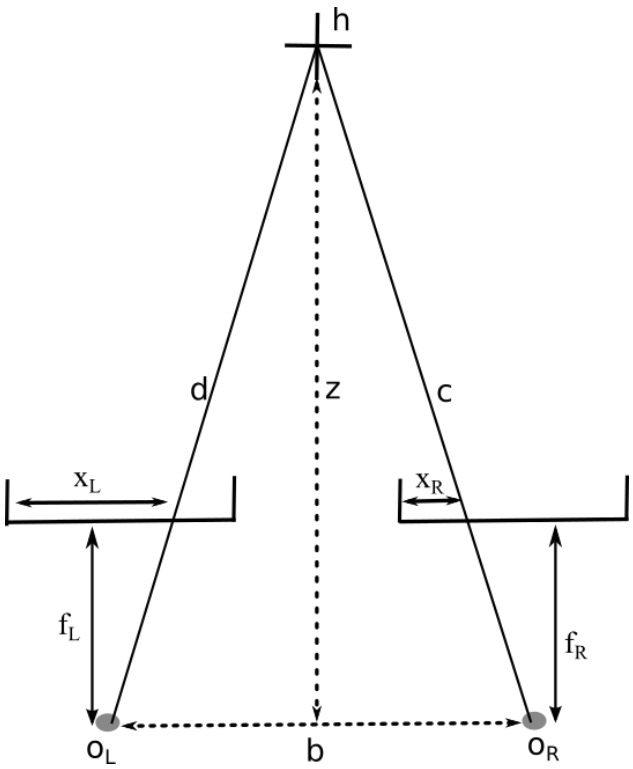


Figure 3.6: Diagram that demonstrates how distance is derived from disparity assuming that the image planes are aligned.

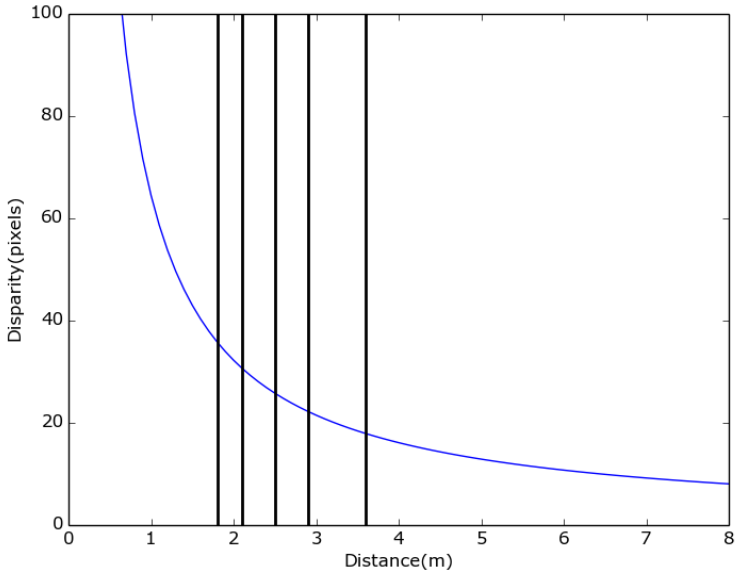


Figure 3.7: Disparity as a function of distance. Black lines represent different accuracies in the disparity levels, the leftmost line represents ± 2.0 disparity level, and the lines following to the right have ± 1.5,± 1.0,± 0.75 and ± 0.5 disparity level accuracy.

The disparity map stage is a computationally heavy operation. Therefore it is important to choose an algorithm that fits well with an FPGA platform. To understand what algorithm we should use, a small overview of existing algorithm was conducted. Stereo vision disparity algorithms have been categorized into four essential steps or substeps [23]:

- Similarity matching, or matching cost computation
- Cost (support) aggregation
- Disparity computation
- Validation and disparity refinement

The first step, similarity matching, is done by comparing pixels in one frame, called the base frame, with the matching stereo frame. Comparing the pixels can be done with squared intensity differences (SD), absolute differences (AD) or more robust methods that help the next step in the algorithm to perform better[23]. The dissimilarity of the pixels is called cost. How we compare the two frames depends on the algorithm, but to reduce the amount of computations performed a *horopter* is established. The horopter sets minimum disparity and maximum disparity between the frames [20]. That horopter is then used to limit the disparity algorithms range in searching for corresponding pixels. The similarity function then creates a *disparity space* which is a cube that has the same size as one frame and a depth that is determined by the horopter [23].

The second step is cost support aggregation. This step is different depending on which particular disparity algorithm is used. Cost support aggregation is essentially the process of finding a potential matching pixel in the disparity space by means of a *supporting region* [23]. The supporting region usually is a window function that compares the neighboring pixels at that disparity with the anchoring window in the reference frame. The anchoring window is not moved while comparing with all the possible matches in the other frame, the range the window travels is set by the horopter. The anchoring window is then moved to the next pixel in the reference frame and same procedure of traversing the range is followed. This step is often skipped by global algorithms, since in the global case the support aggregation is often implicit in the disparity computation [23].

The third step is the disparity computation. How disparity is computed is dependent on the algorithm but they all try to find the matching pixel based on the previous steps. The algorithms can be divided into three groups: local, global, and hierarchical algorithms [23]. Local methods use a window of data centered on the position where the output disparity is to be calculated, see Section 3.6.1. These methods lack accuracy and reliability in texture-less environment but perform with good speed [24]. In contrast, there are global graph methods that use values from all over the frame in a graph structure, propagating information between them to find the disparity of a single pixel. These methods do not generally work in real time but have higher accuracy than the local methods [24]. In between global and local methods we have also hierarchical methods that employ a hybrid of the two

methods, for example block based belief propagation [25]. The hierarchical methods try to reduce the accuracy of a global method to get an increased speed in the computation.

The fourth step is to refine the disparity output. The refinement is often done with median filters that remove spurious disparities which are also known as spikes. Another refinement method is to achieve sub-pixel disparity through curve fitting between the calculated values, the curve is then used to get subpixel data. Also of importance is how *occluded areas* are dealt with. Occluded areas are the areas that one camera sees but the other one does not, e.g. when one camera sees behind a human but the other camera does not see that area. Occluded areas can be dealt with by assigning a likely neighboring value to the area or by using surface fitting techniques [26][23].

In the following subsections we will briefly describe two algorithms that represent local and global methods.

3.6.1 Sum of absolute differences

The traditional method of detecting similarities is with the sum of absolute differences (SAD) [23]. A similarity measure is computed by subtracting the intensities of pixels in one frame from the corresponding intensities in the other frame and then taking the absolute value. The matching cost is then summed up inside a local window. Equation 3.11 describes this process [27].

$$SAD(x,y,d) = \sum_{h=-\lfloor \frac{w}{2} \rfloor}^{\lfloor \frac{w}{2} \rfloor} \sum_{k=-\lfloor \frac{w}{2} \rfloor}^{\lfloor \frac{w}{2} \rfloor} |I_L(x+k, y+h) - I_R(x+k, y+h+d)| \quad (3.11)$$

Here w is the length of the edge of the window; the window is a square and the lengths of its edges are odd, I_L is the left image and I_R is the right image. x and y are the coordinates inside the frames. The window is moved over the images along the same row until it reaches the maximum disparity value. Then the minimum error from Equation 3.11 is chosen as the corresponding pixel in the other image. This optimization of taking the lowest error is called Winner Takes it All (WTA) [23]. The method can introduce errors in disparity when observing texture-less or repeating regions.

Disparity refinement methods can help with cleaning up outliers and occluded areas. Median filters can filter out the outliers. Occluded areas can also be reduced by comparing the right image to the left image as well as comparing the left image to the right image [23]. The last method would shrink the disparity map by the size of the SAD window. Furthermore for most local methods accuracy is an issue [23]. However, if the camera has a small distance that it has analyze, for example a 1.5 m range, then the camera can be set up so that an resolution of few disparity levels does not affect the the distance calculation. This can be seen from Figure 3.7.

3.6.2 Belief propagation

Belief propagation (BP) is a global method that achieves high accuracy disparity. Implementations of the BP as an hierarchical method on an FPGA have shown promising results in regards to running time [25]. However, as a global method on a CPU, it has achieved 1 second processing time for one image pair [28]. BP assigns each pixel with a label f_p in a finite set L. In the case of disparity calculations, f_p represents disparity and L is limited in size by the horopter. The algorithm uses messages that are passed through Markov random field (MRF) models to try to converge on a correct disparity between frames. The algorithm seeks to find the labelling f that minimizes Equation 3.12. The equation gives the energy in the difference between the labelling and the actual disparity. The energy is given by two parameters, as seen in Equation 3.12 [29].

$$E(f) = E_{smooth}(f) + E_{data}(f) \tag{3.12}$$

Here, E_{data} is the difference from the observed data and E_{smooth} measures how much the labeling f is not piecewise smooth [29]. The algorithm is an approximation of the optimal solution since labeling an MRF has been shown to be an NP-hard problem [30]. The algorithm has to iterate over the whole graph many times in order to get accurate results, the intermediate data of the algorithm is the disparity space and hence a lot of data has to be stored in between iteration. The main drawback of belief propagation is then the intensive memory storage requirements needed to process one frame.

4

Implementation

This chapter will describe how the algorithms were implemented in the FPGA, including general system architecture and a description of the interfaces for input, output and memory. The chapter does not aim to give a line-by-line description of the HDL code, but will describe relevant implementation details of the different processing stages. The verification methods used are also described.

4.1 Architecture overview

The system is realized as a series of processing blocks. As the calibration matrices for the remapping are too large to be stored in on-chip RAM, external DDR3 memory is used. Intermediate data is stored in on-chip RAM. The general architecture can be seen in 4.1. As the system is pipelined, the throughput of the system is equal to the throughput of the slowest processing block, while the latency is equal to the latency of the slowest processing block multiplied by the total number of processing blocks. The system clock speed is 200 MHz.

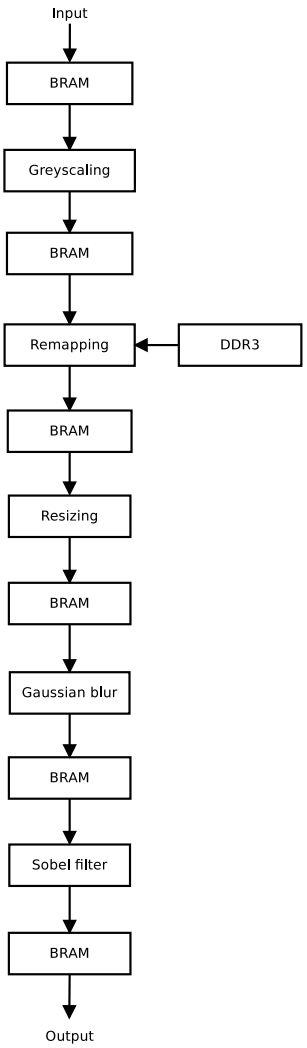


Figure 4.1: Top level block diagram.

4.2 Interfaces

In order to read and write data, store intermediate values, and debug the system, a number of different interfaces to other devices had to be developed. These interfaces are described in this section.

4.2.1 Video input assumptions

At the start of the project, the video input method was not known. Therefore a few assumptions were made about the input. It was assumed that the camera would act as a *rolling shutter* and that it would send one row at a time. Furthermore it was expected that there would be some delay time between the rows and that the input could be fed straight into the greyscaling stage. It was

also assumed that the camera would be connected to the system through an ethernet connection and therefore be limited to a data rate of 1 Gb/s.

4.2.2 Memory

This subsection will briefly describe the forms of memory employed in the system. It only covers the memory expected to be used in a final product during run-time. The verification section briefly describes how additional memory has been used in development and can be used for automatic configuration of the FPGA.

4.2.2.1 Block RAM

The Artix-7 XC7A200T has 13140 kbit of on-chip RAM available. The size of memories created in the design is always a multiple of 18 kbit (other sizes can be chosen, but an amount of resources corresponding to the next multiple will be used). The block RAM is highly configurable, with the width and depth being development parameters. Two blocks can be appended to each other if a larger single memory space is required. The RAM is true dual port, meaning that two read or write operations can be performed at the same time, if care is taken not to work on the same address (which may corrupt data). If so desired, the two ports can use a different clock [31].

In the system, the block RAM is used to implement row buffers for the different stages. Half the memory space is used by one stage to store output data, while the other half is used by the other stage as input data. When both are done with one row, they switch. Later stages, when the frame size has been reduced, employ full frame buffering. This follows the same principle, but with one whole frame being written or read before switching. The dual ports are very useful, as they allow pipelined processing stages to read and write to a common memory space concurrently. As the port width is set by the designer (within some constraints [31]), multiple pixels can be read or written in a single operation.

4.2.2.2 DDR3

The Dual Data Rate 3 (DDR3) memory used to store frames between pipelines was controlled using an IP core generated with Xilinx MIG (Memory Interface Generator) . This core gives a simpler interface compared to working directly towards the memory. It also provides features such as optimizing in which order to read requested memory positions, and then buffering them into the correct order [32]. The controller is clocked at twice the system clock rate, 400 MHz.

The memory itself has a 64-bit wide data bus. As a double data rate memory, it works on both clock flanks, giving a peak transfer rate of 6.4 GB/s. The memory has an $8n$-prefetch structure, meaning that it communicates with the FPGA in bursts of $8 \times 64 = 512$ bits. Data is arranged in columns along rows in four different banks. Initially accessing a row has longer latency than consecutive reads along the same row, motivating certain patterns of memory access. As all the data structures in the system are read and written linearly along rows, the memory can usually be operated at nominal speed.

In addition to the IP controller, an arbiter was written to divide memory access among the different processing stages. The arbiter receives requests from the different processing stages to use the memory, along with command code and possibly data to be written. The arbiter handles requests in a fixed order determined from the memory bandwidth needed by each processing block. This was proven unnecessary in the end, as the abundance of on-chip memory makes the DDR3 needed only by the remapping stage.

4.3 Interstage synchronization

In order to synchronize all stages, there is a need to have a control state machine for each stage. The signals we use are ready signals that propagate either forward or backwards. In Figure 4.2, we demonstrate a connection diagram between three blocks - previous (p), current (c) and next (n). The controller for each stage then follows a state machine like described in Figure 4.3. In this diagram, the two first numbers at each transition shows which value the external stage ready signals should be in order to allow a transition (first the ready signal from the previous block, then the ready signal from the next block). The two numbers to the right of the slash indicate what values the output ready signals to the previous block and the next block should be asserted as when the transition occurs. An x means that the value does not matter. Transitions from the GO state also depend on stage-specific internal logic (for example it only transitions to input wait when the calculations on one row are complete).

If the input double buffer and the output double buffer are of the same size, then handshakes are made with both the preceding and the succeeding stage before calculations start. At the conclusion of a handshake, the two involved subsystems switch which area of the shared RAM that they work on. However, the system switches from buffering rows to buffering whole frames after the resizing stage, which requires a special solution. This is simply handled by stage-specific logic ensuring that handshakes with the Gaussian blur are only made after a certain number of handshakes with the resizing.

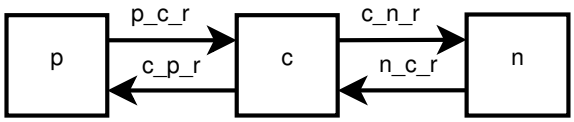


Figure 4.2: Synchronization signals between processing stages.

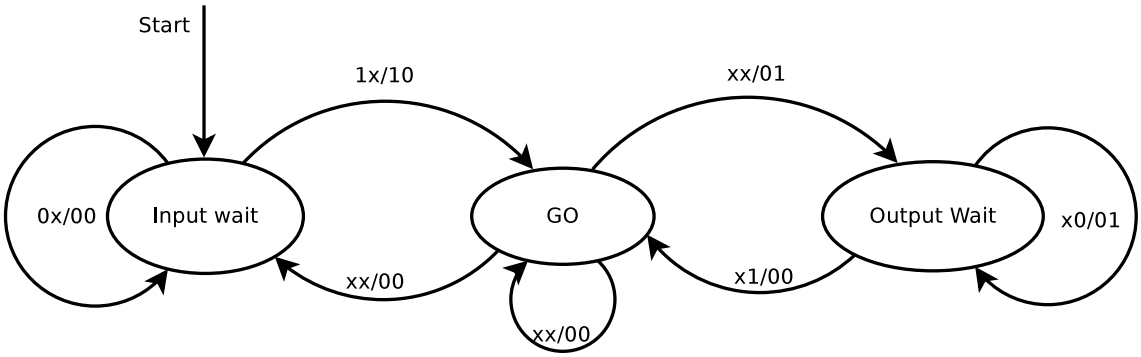


Figure 4.3: Synchronization state machine.

4.4 Implementation details

This subsection summarizes how the processing described in the theory section was implemented. All calculations on the amount of clock cycles needed for a stage are assuming an ideal case where the throughput of the system is not limited by the frame rate of the video source.

The block diagrams in this section are color-coded in the following manner:

White blocks Entities directly used by the processing stage illustrated.

Red blocks Neighbouring entities, included to make the data path clearer.

Grey lines Synchronization signals.

Blue lines Image data.

Red lines Other data, such as coefficients stored in memory.

4.4.1 Greyscaling

This processing stage has input pixels with a length of 24 bits, eight each for the red, green and blue component. The output pixels — as well as the weights used to calculate them — are eight bits long. The greyscaling needs three multipliers for each pixel it should be able to calculate in a clock cycle. The input memory is treated as a double row buffer.

Due to limitations in the interface between Microblaze and BRAM, the write port width of the input buffer was limited to powers of two. Due to this limitation, some additional logic to sort data read into the greyscaler correctly was added. For example, if 256 bits are read in one clock cycle during operation, only 192 bits are processed. The remaining 64 bits are stored in a register, and it is processed in the second clock cycle, with the first 128 bits of the second memory read appended to it. The remaining 128 bits are used in the third clock cycle with the first 64 bits of the third memory read appended to it. The remaining 192 bits are stored and processed in the fourth clock cycle, and the process then repeats itself. This means that no memory is read in the fourth clock cycle. A simplified block diagram illustrating the greyscaling component is shown in Figure 4.4.

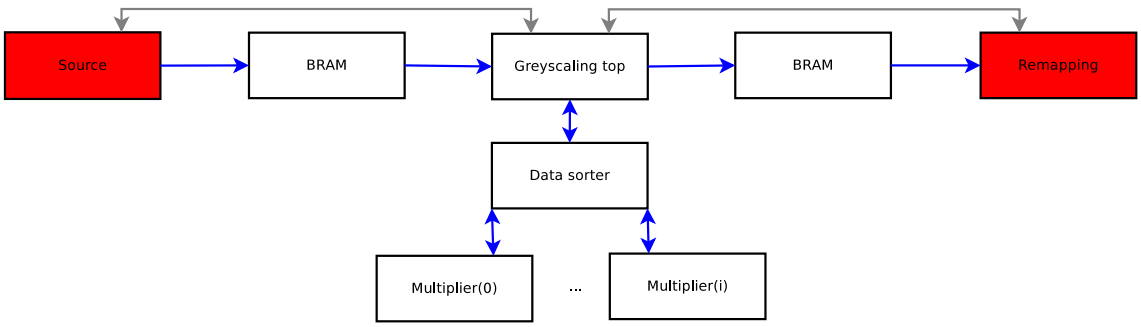


Figure 4.4: Block diagram of the greyscaling implementation.

4.4.2 Remapping

Remapping is the next stage in the image processing pipeline. Remapping shares a block RAM with the greyscaling where data is buffered and synchronization signals (described in Section 4.3) are used to control access to it (see Figure 4.5). Implementation of this stage is dependent on parameters from the commercial Bumblebee stereo camera [9].

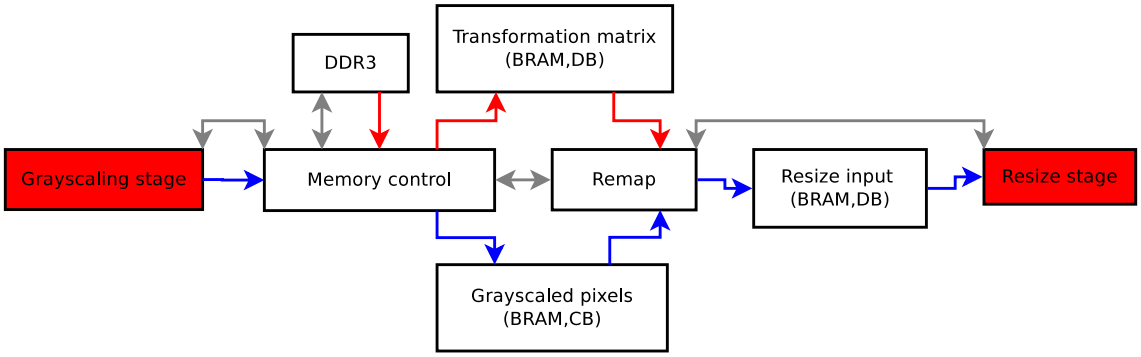


Figure 4.5: Top level block diagram of the remap stage implementation.

The top level block view seen in Figure 4.5 can be divided into three main areas: memory controller, remap and interpolator. The memory controller's role is to handle the access to the input memory and to make sure that the greyscaling is not overwriting data that has yet to be used by the remapping stage. Remap fetches input pixel data from the input buffer, pairs it with the correct transformation weights and signals the interpolator to work on the data. The interpolator then takes the data into a pipeline where the data is processed and stored into the double buffer in the resizing stage.

Most stages in the pipeline have a double row buffer, but the buffer between greyscaling and remapping stage was chosen to be a circular row buffer(CB). The reason for choosing a CB is because each output row might need multiple rows from the input. The memory controller lets each stage have its row in the CB and the stages have to ask permission to start on the next row. The amount of rows needed to store on the input in the circular buffer depends on the transformation matrix values which are computed in a foreground calibration.

4.4.2.1 Memory controller

The memory controller reads pixels into the CB and and weights into the transformation matrix double buffer (DB). On start-up, the memory controller reads in a row from the DDR3 external memory and compares all the transformation weights to find the highest and lowest row needed in the input memory. That is, the highest row is the row closest to the top of a frame and the lowest is the row closest to the bottom of the frame. It then reads continuously from the greyscaling stage until the lowest row needed for that output frame row has been written into memory, the memory controller then signals the remapping stage that all input data is ready. The next transformation matrix row in the DDR3 is then read simultaneously as next row of the input greyscaled pixels is read into CB memory.

The memory controller stalls the input when it reaches the highest row that is being processed by the remapping stage and the remapping stage stalls until it gets the lowest row. Therefore the memory has to be bigger than the largest distortion in the camera and a little bit more if possible since the remapping stage does not process every row at a constant speed and because of the streaming input we might need to keep a continuous input pixel flow. The input memory requirement of distortion from the example stereo camera can be seen in Figure 4.6. The figure is plotted by examining the highest and the lowest row on the input needed for that row on the output. That difference is then the input storage need of that transformation matrix. As can be seen, the requirements for storing input rows are greater around the edges, since the distortion is greater closer to the edge of a frame.

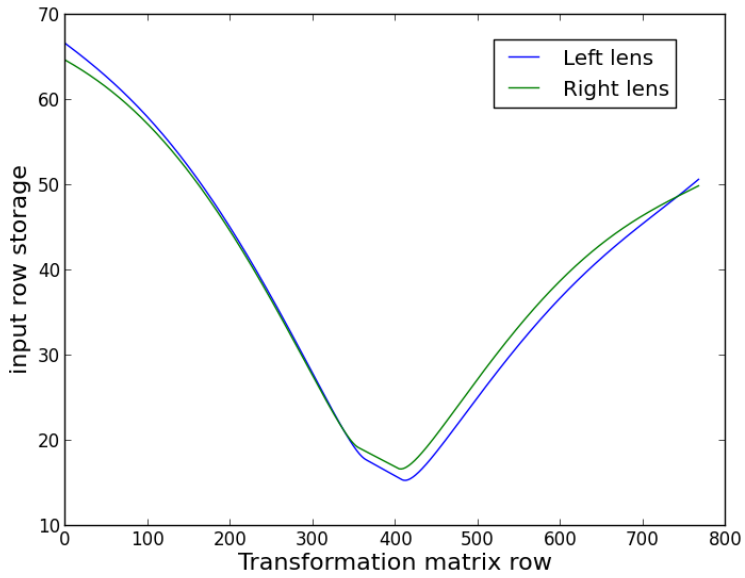


Figure 4.6: The amount of lines needed to buffer at the input as a function of a row in transformation matrix.

4.4.2.2 Remapping

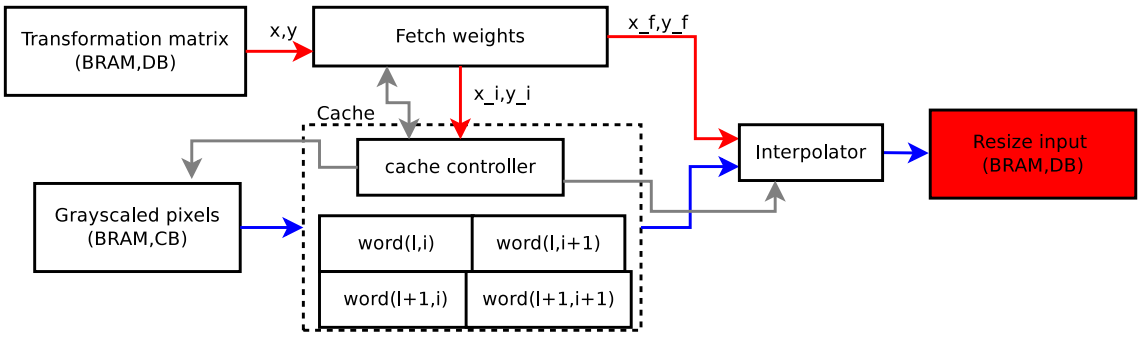


Figure 4.7: Block view of the Remap stage.

In Figure 4.7 we can see the remap stage; it consists of the fetch weights stage (FW), cache controller (CC) and an interpolator. The FW stage interacts with the memory controller and handles the synchronization logic. When FW reads the first transformation weights, it checks if the input has reached the lowest row for that output pixel row. If the input has not reached that row, then the FW stalls until that row is in memory. When all rows are in memory, the FW stage tells CC that the input for that row is ready and sends CC the first transformation weights. CC then reads in the pixels needed for the transformation weight and stores them in block RAM access registers. The CC reads a memory element into the cache which is called word, word is the size of the port width to memory and includes a fixed number of greyscaled pixels. When the registers have been updated the CC signals the interpolator that it has all the data and ask FW for the next transformation weights.

The size of the transformation weights has a critical effect on the off-chip memory bandwidth. Equation 4.1 tells us how much off-chip bandwidth in MB is needed for the stage:

$$M(f_s) = 2f_s(HW)(2(I+F)) \tag{4.1}$$

Here H is height of frame, W is width of frame, f_s is stereo frames per second, I is the size in bits of the integer part of the weights and F is the size in bits of the fractional part of the weights. In the implementation described here, I is 10 bits and F is 16 bits. Now, we put all the known factors in the equation above we get that for each stereo frame processed per second we need memory bandwidth of $M(f_s) = f_s 10.2MB/s$. If the remapping is the only stage needing to access to external memory, 626 transformation matrices could be read from memory per second.

CC was implemented to reduce the block RAM memory access since we always need pixels from two different places in memory if the port width is two. The amount of pixels to be fetched from one frame row in the block RAM was decided from Figure 4.8. Eight column pixels from the same row was chosen as the fetch size, since it was the lowest average fetch size for the entire transformation matrix in both lenses. The port width for the current transformation matrix is investigated with different port widths in Section 5.2.

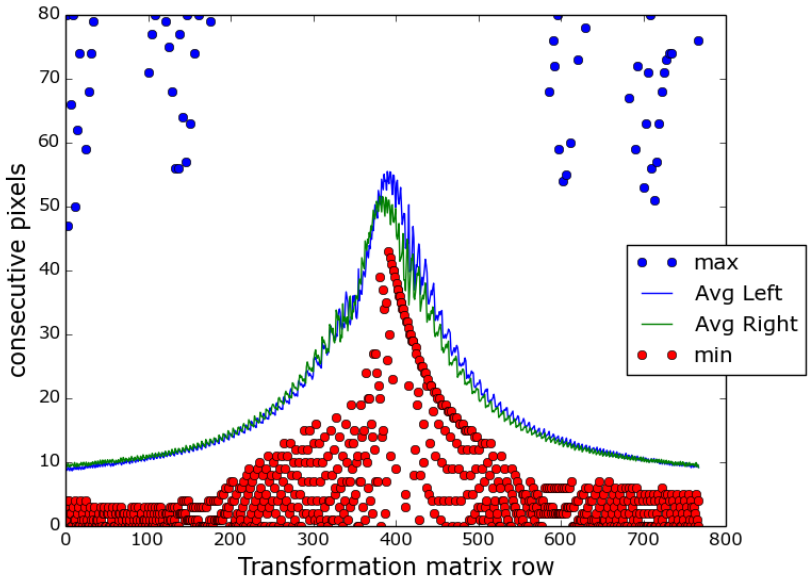


Figure 4.8: Statistics on how many consecutively column pixels are fetched from the same input frame row.

The set of accesses needed from the input block RAM can be seen in Figure 4.9. Black pixels in Figure 4.9 are pixels that are not in the cache, white pixels are in the cache, red pixels are pixels that need to be fetched from the input block RAM into the cache, o is that starting transformation weight, and x is the next transformation weight.

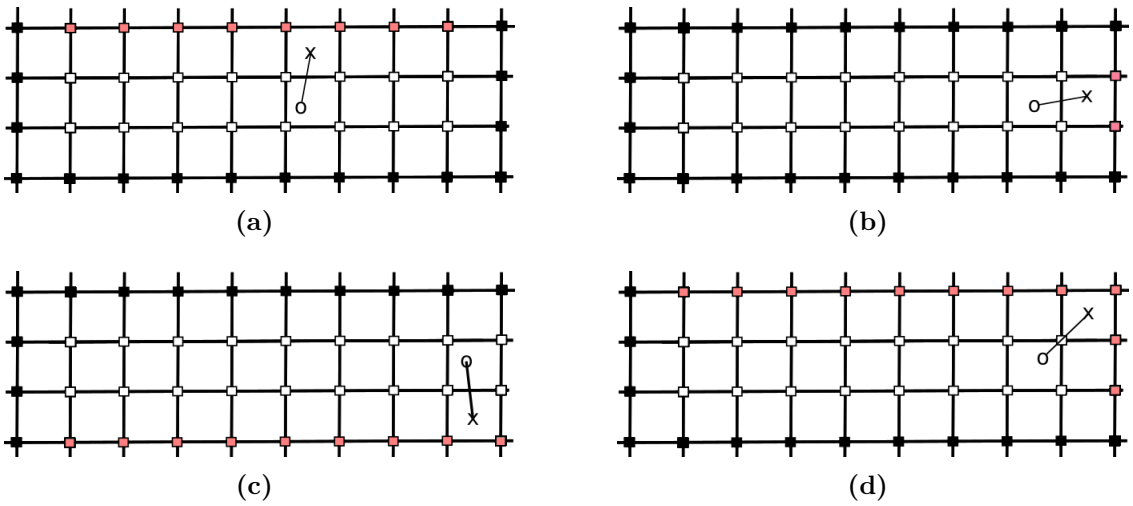


Figure 4.9: A demonstration of the four types of input pixel row fetches that the cache controller performs.

When the stage starts a new output pixel row, it loads into word (l, i) and word $(l + 1, i)$ (see Figure 4.7) the pixels that are needed for the first calculation, where i is the column address in block RAM and l is the row address in block RAM. It then

continues until the next transformation weight points to a pixel that is not in the cache. If the input pixel needed is the one above or below, then one word read will suffice. An example of a fetch that fetches the word above can be seen in Figure 4.9(a). If we need the row above then cache word $(l+1, i)$ is used to store the input pixel values in cache word (l, i) and cache word (l, i) is use to store input pixels in the word address above, that is, word $(l-1, i)$.

If the same row is still needed when we get to the last column pixel stored in the cache, we need to do a *right shift*. Such a fetch is demonstrated in Figure 4.9(b). Word (l, i) and word $(l+1, i)$ are in memory and the word pair word $(l, i+1)$ and word $(l+1, i+1)$ are also needed in order to get input pixels that are across block RAM address boundaries. Then two memory fetches are performed and all four words are kept in memory.

If it happens that we are on the address boundary and the next transformation weight is above or below, then we have the fetch case demonstrated in 4.9(c). Two fetches are needed, one word below to the right and a word below to the left. The left fetch is quite wasteful since we fetch 8 pixels for just the last pixel. Gribbon et al. suggest[33] to use instead a three point bilinear interpolation to save us the trouble of getting that corner pixel but in this implementation we still do that fetch.

The last and the worst case fetch is when pixels need be fetched across block RAM address boundaries and at the same time we need a word in the row above or below. This is demonstrated for the row above in Figure 4.9(d). Then we need fetch from three memory locations, word $(l-1, i)$, word $(l-1, i+1)$ and word $(l, i+1)$. The same applies as in the fetch before that we only need one pixel from the word $(l-1, i)$.

4.4.2.3 Interpolator

The interpolator is described in Section 3.3 by Equation 3.3. We implement that equation by splitting it into four parts (U(1), U(2), U(3), U(4)). If we add these parts together we should get the desired output pixel.

Table 4.1: Expansion of different components from equation 3.3.

$U = U(1) + U(2) + U(3) + U(4)$			
U_1=	$a(1-x)(1-y)$	$a(1-x-y+xy)$	$a((1-y)(1-x))$
U_2=	$bx(1-y)$	$b(x-xy)$	$b(x-xy)$
U_3=	$cy(1-x)$	$c(y-xy)$	$c(y-xy)$
U_4=	$d(xy)$	$d(xy)$	$d(xy)$
Expansion	1	2	3
Additions	3+2	3+5	3+4
Multiplications	4+4	4+1	4+2
Latency	5	6	5

The addition of U(1), U(2), U(3) and U(4) to produce the output pixel and the multiplication of the pixels (a,b,c,d) with the transformation fractional part (x,y) cannot be reduced, but the operation of multiplying the transformation fractions together can be reduced. In Table 4.1, three different expansion from the original

equation are investigated. Multiplication and addition have the constants 3 and 4 respectively to show the operations that cannot be expanded. Multiplication is a more power-expensive operation than addition; therefore we seek to reduce the multiplication done by the interpolator. But as we can see in Table 4.1, latency can also become an issue, since in expansion 2, we have the fewest multiplications but we have to store all the pixels and transformation fractions for one clock cycle extra. Therefore, expansion 3 was chosen to be implemented, since it had the lowest latency and number of multipliers.

Table 4.2: Operations done in each stage of the Interpolator.

stage	1	2	3	4	5
Op.	1-y	(1-y)(1-x)	$a((1-y)(1-x))$	U(1)+U(2)	U(1,2)+U(3,4)
	1-x	x-xy	$b(x-xy)$	U(3)+U(4)	
	xy	y-xy	$c(y-xy)$		
			$d(xy)$		

To be able to clock the interpolation at a high rate, we need to have multiple pipeline stages. The amount of pipeline stages affect the latency and the power consumption. Table 4.2 shows the calculation done in each stage in the interpolator.

4.4.3 Resizing

The resizing scales down an image frame from 1024 by 768 pixels to 307 by 230 pixels in order to decrease the amount of calculations needed later in the signal chain. The resolution is based on the software provided by the advisor - higher resolution could potentially be used in a final hardware system. The ratio between the input and output frame is illustrated in Figure 4.10. It is performed in two steps, first along columns and then along rows. Weights as well as what input pixel indices should be used for each output pixel is calculated offline and stored in read-only memory on the FPGA at configuration. The weight window is nine pixels wide. Input pixels, output pixels and weights are all eight bits long. The input memory works as a double row buffer, identical to the greyscaling. The intermediate frame data between column reduction and row reduction is stored in a double frame buffer, requiring a significant amount of the available BRAM.

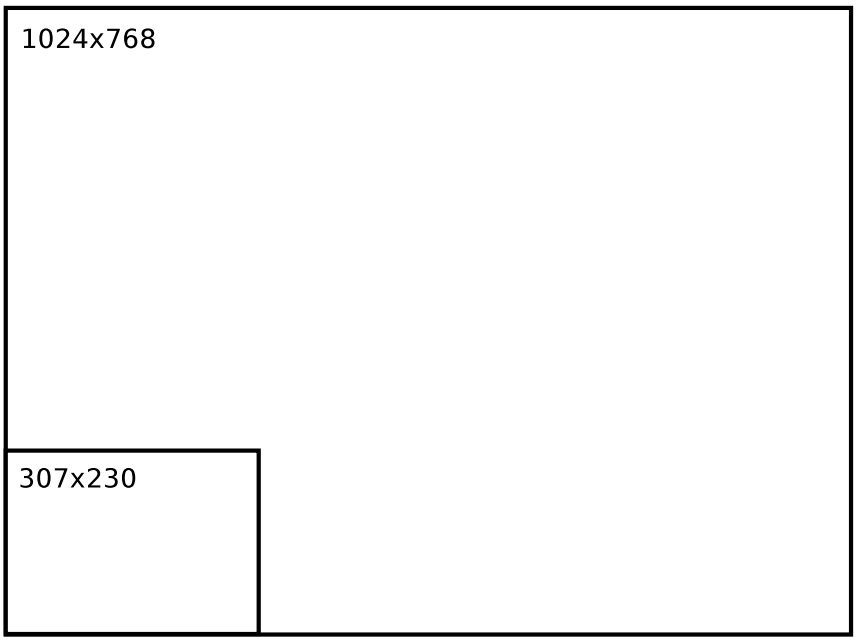


Figure 4.10: Ratio between original and downscaled frame.

As can be seen, the resizing is in essence two largely identical processing stages, one working horizontally and one working vertically. In each part, a top level handles IO and synchronization. Beneath this level, a sorting unit is instantiated, which routes pixel data to averaging units as specified by the contents of the index ROM. Weights are taken from the weight ROM. The averaging units are instantiated below the sorter, and finally multipliers are instantiated below each averaging unit. Each averaging unit uses nine multipliers. One averaging unit is needed for each output pixel to be calculated in parallel in each step of the resizing.

The resizer reads new data from the input BRAM when signalled by control logic in the sorter. This logic checks if the index ROM indicates that there are pixels needed in the next output calculation that are not among the currently available data. As the windows of the different output pixels may overlap, only part of the data is changed during this operation. Specifically, only a third of the data cached inside the sorter is changed. The remaining data is shifted and the current base index of the cached data is updated. A total of 24 pixels are cached in the column reducing section at any given time, while a total of 18 pixels are cached in the row reducing section. Pipelining is used in order to meet timing constraints - data is stored after input, after multiplication, and after the partial results are added together into the output pixel.

The port width from the input memory to the column reduction is 512 bits, corresponding to 64 pixels. The ports of the intermediate memory and the output memory, however, are only one pixel wide. Smaller port widths were used since the first stage outputs along rows, while the second reads along columns. Using wider port widths to increase throughput is possible, but would make the memory control more complex.

A simplified block diagram of the resizing is shown in Figure 4.11.

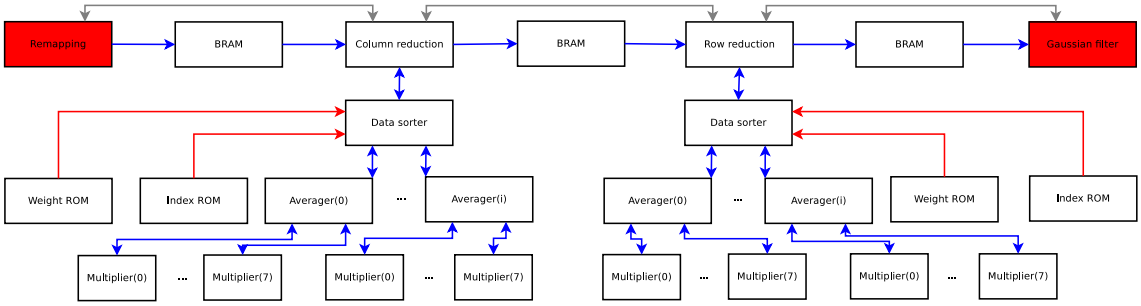


Figure 4.11: Block diagram of the resizing implementation.

4.4.4 Gaussian blur

As the window used is seven by seven pixels wide, 49 multiplications and 48 additions are required to calculate one output pixel. Similar to some of the preceding stages, the blur is implemented using a top level that handles reading, writing and synchronization. Below the top level, there is a sorting entity that routes data between the top level and the convolution entities. The convolution units themselves are instantiated below the sorting unit, and finally individual multipliers are instantiated below the convolution units. The window moves from left to right along the top row, and then along the next row in the same manner, until the entire output frame has been calculated. It follows that 49 pixels have to be read from memory at the start of a row, but later pixels in the same row only need seven new pixels made available. Input data is stored in a double frame buffer.

As only 49 weights are needed, they are stored in look-up tables rather than BRAM in order to slightly simplify the code. Weights, input pixels and output pixels all use a word length of eight bits.

A simplified block diagram of the Gaussian blur is shown in Figure 4.12.

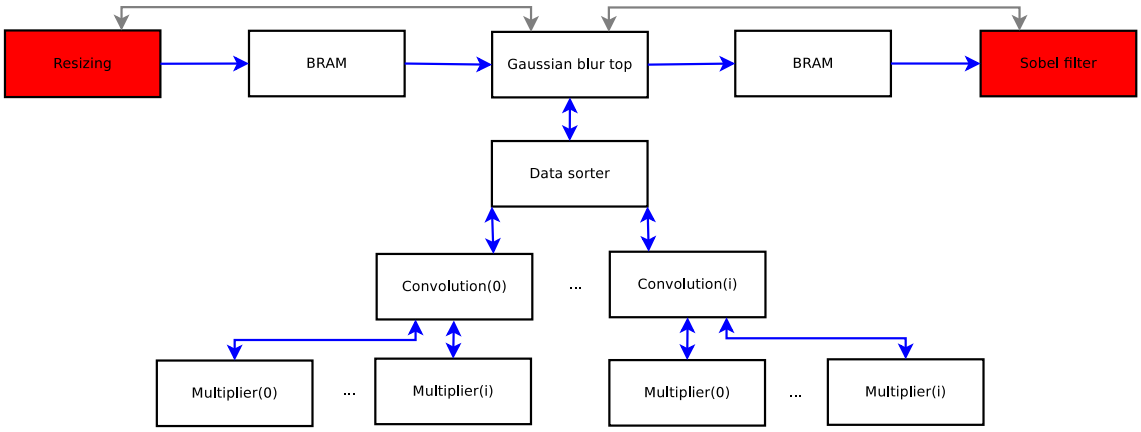


Figure 4.12: Block diagram of the Gaussian blur implementation.

4.4.5 Sobel filter

The Sobel filter enhances edges as described in Chapter 3. In the implementation, the convolutions are not performed using multipliers. Instead, all input pixels are

extended to a signed 16 bit representation, with the pixels to be subtracted given a negative sign. The values that need to be multiplied by 2 are left-shifted one bit. The values are then added together. This is done for both the x-axis kernel and the y-axis kernel concurrently. Input data is stored in a double frame buffer.

The filter is structurally similar to earlier processing stages, as is illustrated in Figure 4.13. No multipliers are instantiated, however, as mentioned above. The zero-padding and shifting is performed in the convolution entity. Memory accesses are handled in the same manner as for the Gaussian blur, but scaled down to a three by three window.

In order to decrease complexity, the magnitude of the gradient is not calculated using a square root. Instead, it is approximated as the sum of the absolute value of the x- and y components (see Equation 4.2, where G is the gradient, G_x its component along the x-axis and G_y its component along the y-axis). This approximation gives correct values when the gradient is parallel to an axis, but does not give perfect results at other angles.

$$|G| \approx |G_x| + |G_y| \tag{4.2}$$

The filter is susceptible to overflow. This is handled through saturation. This results in somewhat thicker edges than in the ideal case.

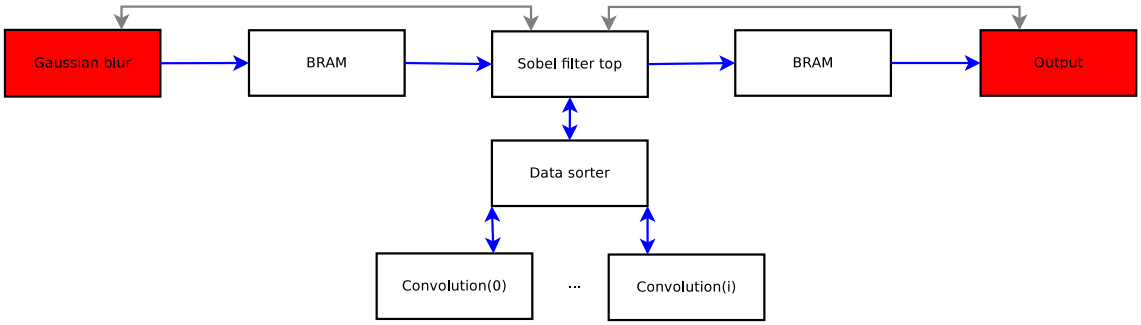


Figure 4.13: Block diagram of the Sobel filter implementation.

4.5 Verification

A number of different tools were used for verification during the development. Initial evaluations were done with testbenches in the Vivado simulator in order to ensure basic functionality. Tests were then performed on-chip, using the On-Chip verification as described below. Vivado supports use of integrated logic analyzers (ILA) on-chip in order to capture signals when certain conditions are met and then transfer them to the PC over JTAG. This was particularly useful when doing development relying on hardware external to the FPGA, such as the DDR3 memory.

The Python packages Numpy and OpenCV were used to convert PNG images into binary matrices with RGB data in a row-major order, suitable for writing to the Flash memory on the development board in order to perform on-chip verification without a real-time video feed. Python scripts were used to convert output data into PNG images as well.

A simplified block diagram of the verification environment is shown in Figure 4.14. A more detailed description on the development board and the FPGA is described in Section 1.4.1.

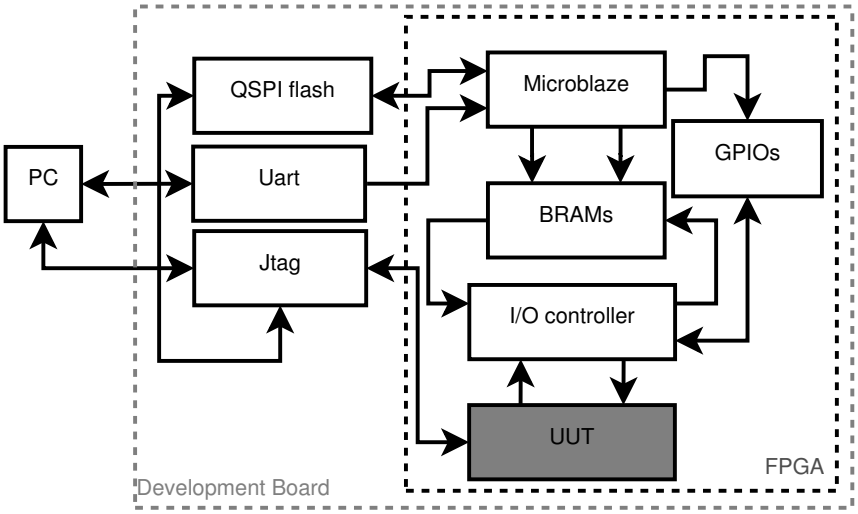


Figure 4.14: Block diagram of the verification environment.

4.5.1 On-Chip functional verification

To verify functionality of each stage, two stereoscopic images are run through the system for quality and integrity comparison on the output with a software solution. The datapath of the verification is seen in Figure 4.15.

Before on-chip functional verification can start, we store the stereoscopic images in the off-chip flash memory. We then let a Microblaze [34] read from the flash and then store one frame row in the input BRAM. When the Microblaze has finished writing one row to the input BRAM, it signals the I/O controller to read the frame row, through the Xilinx IP AXI GPIO. The controller then sends the row to the unit under test (UUT). The UUT then works on one row or waits for more rows from I/O controller, depending on the input requirements of the UUT. We then record the execution time of one row with ILA and can estimate the theoretical speed of the stage, i.e. the number of frames the stage can execute per second without any data bottleneck that might occur in other stages such as the camera feed, output of data, or any other stage of the pipeline. The UUT send the image frame data back to the I/O controller which stores the results in an output BRAM (both memories are called BRAMs collectively in Figure 4.14). When the output BRAM is full, the data is read by the Microblaze and sent over the UART to the PC conducting the test. There the output frame is compared with already processed frame, to ensure that the system does not have any errors.

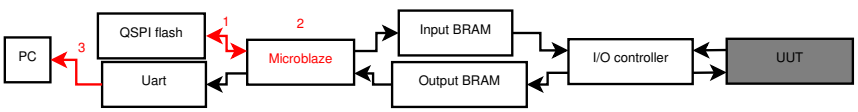


Figure 4.15: The datapath for the verification system. The numbers in the figure represent bottlenecks that inhibit faster processing speeds.

The reason that we cannot get a reasonable time measurement for a whole frame through the device is the bottlenecks in the data communications (as will be discussed below) and the available BRAM in the system. A whole input test frame cannot be stored on the onboard BRAM because one frame is larger than the available memory (onboard BRAM is approximately 1.6MB and one image frame is 2.4MB). There are three main bottlenecks in the system and they are colored with red and numbered in Figure 4.15. The first data bottleneck is the QSPI flash interface (number 1 in Figure 4.15). The flash interface can not provide more than 54Mb/s. The second bottleneck is the Microblaze, since it can only read and write to each BRAM separately. The third and final bottleneck is the UART, since the top baud rate is 115kb for our PCs.

4.5.2 Problems with verification

Initially, when planning this stage, we thought it should take three weeks, but because of unforeseen problems it took significantly longer. Initially, we decided that we needed the on-chip functional verification to verify our implementation. We needed the access to the external flash and the testing PC via UART, but we also wanted to be able to have a video interface to test the real time constraints of the system and some possible corner cases. Therefore, we choose to use the Microblaze softprocessor to interface all the external components. Xilinx free IPs all interface together via the AXI bus, and therefore an easy solution was to use the Microblaze to control all these AXI bus communications.

The first problem encountered with AXI system was how to interface the external flash with the Microblaze. There was not any good documentation on the Microblaze drivers and therefore we needed to do a lot more testing than expected. The next problem was how to interface Microblaze with our hardware design. After a bit of searching we decided to use a double buffer method, where we would fill a buffer, and then signal our hardware with a GPIO that data is ready to be read and then write the next buffer. The buffer interface was implemented with an AXI BRAM controller and the signaling was done with an AXI GPIO interface to the AXI bus. Now that our interface method was complete we needed to have VHDL code that handle the communication with Microblaze BRAMs and GPIOs. Initially, we thought that this would be easy, but it turned out to be more problematic than expected. The Microblaze could not be clocked as fast as the rest of the system, leading to two different clock domains. This required a more robust handshaking procedure. Before we started, we knew that the speed of the UART communication and the QSPI flash would slow done the testing procedure. Therefore, we would have delays both when inputting data and on the output. The control layer had to work with sink and source driven methods to handle the delays, which made the handshake procedures more complex.

The last and most troublesome part was getting Microblaze and the controller to work together. ILA was an important tool for solving that problem. We had many kinds of strange behavior from Microblaze, which resulted from the Xilinx AXI interconnect IP module. Apparently, when interfacing native BRAM you cannot use interconnect FIFOs, since that is not supported. Using them caused problems

with data delivery and many tests were performed before the problem was identified. A additional problem the interconnect was the addressing used by the Xilinx IP AXI BRAM controller (ABC). The ABC would just access every fourth word in memory and did not access any in between; we saw that we read and wrote four times to the same memory location. This was solved with counting the write enables and read enables from the ABC and use that counter to address the memory space instead of the controller address. This solution can hardly be considered ideal.

5
Results

This chapter describes the performance achieved with the different processing stages as well as the system as a whole, and comparison is made with software implementations. Images processed by the system are included to illustrate the functionality of the system. Speeds given for individual stages assume ideal conditions, that is, no delays introduced by waiting for a camera or other processing. A frame size of 1024 by 768 pixels is used up to and including the resizing, after which a frame size of 307 by 230 pixels is used.

Power estimations were performed with switching activity recorded in simulations with real images and a clock frequency of 200 MHz. This is also the assumed clock speed in speed estimations. Note that frames per second here refers to frame *pairs* per second, as the intended usage is with a stereo camera.

For comparison, the frame frequency is also given for software implementations. The platform used in the comparisons was an Intel Core i7-4910MQ CPU clocked at 2.9 GHz. Measurements were made both for the single core and the multicore case. The software and measurements of it were provided by the project advisor [35]. It was written in C++ using the OpenCV library.

Utilization given in this chapter includes resources used by the processing stage itself as well as its input BRAM.

The images shown up to the resizing are shown at a smaller size than used in the system in order to fit them into the document.

5.1 Greyscaling

The greyscaling implementation was evaluated for four different cases, enabling concurrent processing of 1, 2, 4, and 8 output pixels. The design is pipelined in such a manner that one batch of output pixels can be output each clock cycle while processing a row. Each row needs an additional seven clock cycles for synchronization. The results for the four different implementations can be seen in Table 5.1, where the period given is for one frame pair. The frame rate for a software implementation is shown in Table 5.2.

Table 5.1: Results for the greyscaling stage.

	Speed		
Concurrent output pixels	Period (cycles)	Period (ns)	Frames/second
1	791,808	3,959,040	126.3
2	398,592	1,992,960	250.8
4	201,216	1,006,080	497
8	103,680	518,400	964.5
	Power		
Concurrent output pixels	Static (W)	Dynamic (W)	Total (W)
1	0.131	0.088	0.219
2	0.131	0.099	0.231
4	0.131	0.112	0.244
8	0.132	0.180	0.312
	Utilization		
Concurrent output pixels	LUT	FF	BRAM
1	1,061 (0.79%)	861 (0.32%)	270 kbit (2.05%)
2	1,131 (0.84%)	928 (0.34%)	270 kbit (2.05%)
4	1,289 (0.96%)	1,068 (0.39%)	270 kbit (2.05%)
8	1,521 (1.13%)	1,093 (0.4%)	270 kbit (2.05%)

Table 5.2: Software frame rate for greyscaling.

Platform	Frames/second
CPU (single core)	608.1
CPU (multicore)	1,460.6

Input and output with an example image is shown in Figure 5.1 and Figure 5.2.

Figure 5.1: Before greyscaling.

Figure 5.2: After greyscaling.

5.2 Remapping

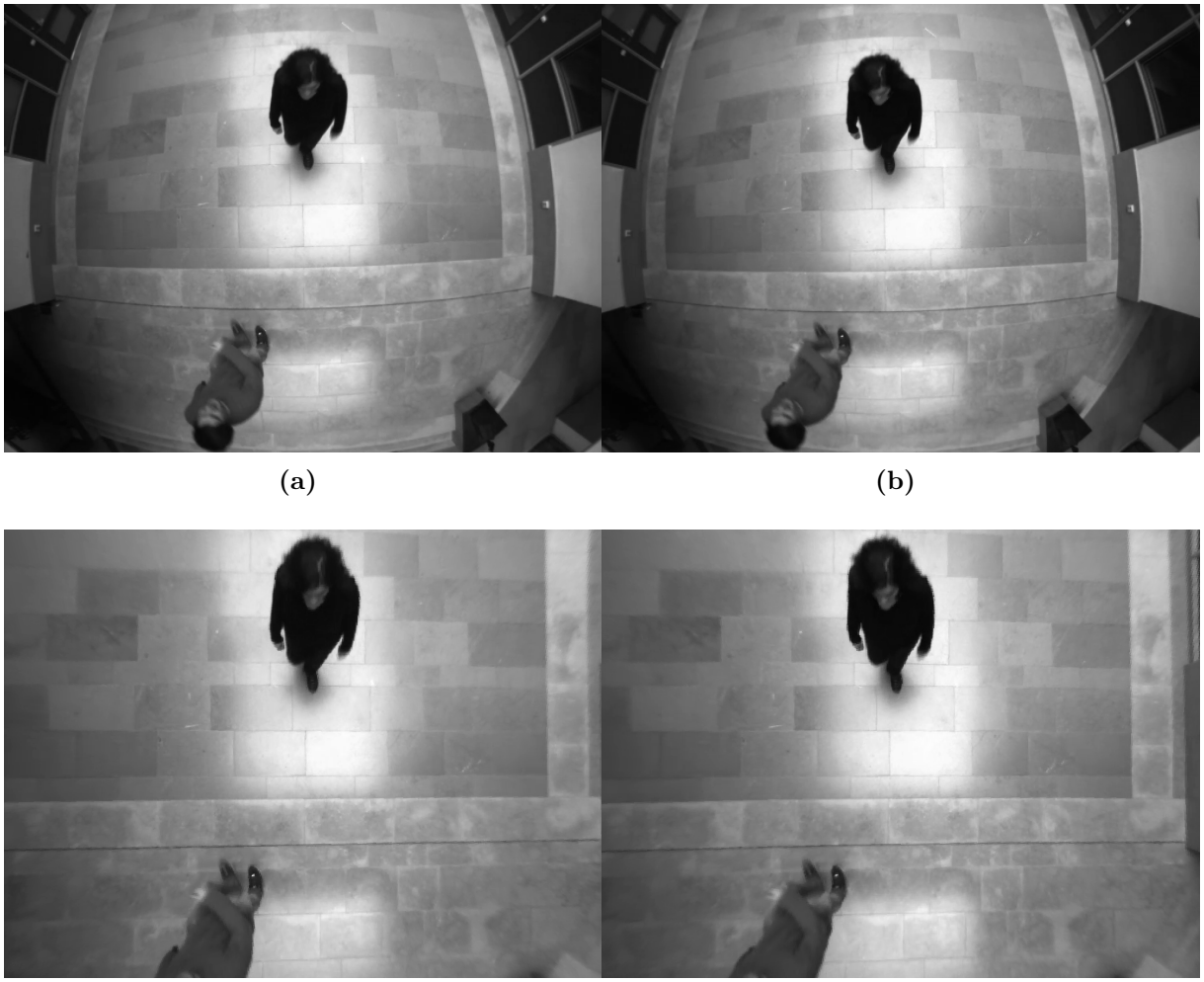
In Figure 5.3, we can see the hardware results for demonstration. As previously pointed out, the stereo frame pair after remapping does not cover the whole visual field that was present in the frames before remapping. Therefore if the misalignment of the frames is great we will have smaller visual field to work with. The remapping stage was evaluated for one pipeline case. The results of our implementation can be seen in Table 5.3. Different fetch width for the cache word were tried out. The remapping stage uses 6 Digital Signal Processing (DSP) slices (0.81%), and it is the only stage that uses such resources. The memory bandwidth needed to DDR3 to sustain the speed of 64.4 Frames/second is 658MB/s, calculated from Equation 4.1. The required bandwidth is 10% of the maximum bandwidth available.

Table 5.3: Results from the remapping stage.

	Speed		
Port width (pixels)	Period (cycles)	Period (ns)	Frames/second
16	3,017,918	15,089,590	66.27
8	3,108,008	15,540,040	64.4
4	3,622,734	18,113,670	55.21
2	4,660,904	15,540,040	42.91
	Power		
Port width (pixels)	Static (W)	Dynamic (W)	Total (W)
16	0.133	0.152	0.285
8	0.133	0.122	0.255
4	0.132	0.133	0.264
2	0.133	0.194	0.327
	Utilization		
Port width (pixels)	LUT	FF	BRAM
16	2,578 (1.92%)	2,840 (1.06%)	1,180 kbit (9.86%)
8	1,964 (1.46%)	2,577 (0.96%)	1,180 kbit (9.86%)
4	1,728 (1.29%)	2,451 (0.91%)	1,180 kbit (9.86%)
2	1,584 (1.18%)	2,389 (0.89%)	1,180 kbit (9.86%)

Table 5.4: Software frame rate for remapping.

Platform	Frames/second
CPU (single core)	133.638
CPU (multicore)	414.364



(a) **(b)**

(c) **(d)**

Figure 5.3: Figures (a) and (b) show left and right remapping input, Figures (c) and (d) are the left and right images after remapping.

5.3 Resizing

The resizing implementation was evaluated for three different cases: using 1, 2, or 4 averaging units for each reduction step. The results are shown in Table 5.5. Software comparisons are given in Table 5.6.

Table 5.5: Results for the resizing.

	Speed		
Averaging units	Period (cycles)	Period (ns)	Frames/second
2	988,881	4,944,405	101.1
4	835,285	4,176,425	119.8
8	758,517	3,792,585	131.8
Averaging units	Power		
	Static (W)	Dynamic (W)	Total (W)
2	0.136	0.317	0.452
4	0.136	0.469	0.605
8	0.137	0.732	0.869
Averaging units	Utilization		
	LUT	FF	BRAM
2	7,366 (5.5%)	7,382 2.75%)	4,320 kbit (32.88%)
4	11,664 (8.71%)	7,788 (2.91%)	4,320 kbit (32.88%)
8	19,422 (14.51%)	7,788 (2.31%)	4,320 kbit (32.88%)

Table 5.6: Software frame rate for resizing.

Platform	Frames/second
CPU (single core)	1,944.8
CPU (multicore)	1,906.9

Example input and output are shown in Figures 5.4 and 5.5.

Figure 5.4: Before resizing.

Figure 5.5: After resizing.

5.4 Gaussian blur

Three different degrees of parallelization were evaluated for the Gaussian blur. These consisted of using 1, 2, or 4 convolution units. The throughput is limited by the BRAM port width used: a maximum of one pixel per clock cycle can be read. For this reason, the differences between the test cases are very small. The results are given in Table 5.7. Software comparisons are given in Table 5.8.

Table 5.7: Results for the Gaussian blur.

Speed			
Convolution units	Period (cycles)	Period (ns)	Frames/second
1	793,036	3,965,180	126.1
2	757,845	3,789,225	132
4	740,136	3,700,680	135
Power			
Convolution units	Static (W)	Dynamic (W)	Total (W)
1	0.133	0.072	0.205
2	0.133	0.112	0.245
4	0.133	0.207	0.34
Utilization			
Convolution units	LUT	FF	BRAM
1	1,573 (1.17%)	2,127 (0.79%)	1,332 kbit (10.14%)
2	2,247 (1.67%)	2,743 (1.02%)	1,332 kbit (10.14%)
4	3,712 (2.77%)	3,903 (1.45%)	1,332 kbit (10.14%)

Table 5.8: Software frame rate for Gaussian blur.

Platform	Frames/second
CPU (single core)	3,327.7
CPU (multicore)	2,141.8

Example output is given in Figure 5.6.

Figure 5.6: After Gaussian blur.

5.5 Sobel filter

Results for the Sobel filter are given here. It suffers from the same limitations as the Gaussian. The processing speed, power consumption, and utilization can be seen in Table 5.9. Software comparisons are given in Table 5.10.

Table 5.9: Results for the Sobel filter.

Speed	Period (cycles)	Period (ns)	Frames/second
	499,556	2,497,780	200.2
Power	Static (W)	Dynamic (W)	Total (W)
	0.132	0.031	0.163
Utilization	LUT	FF	BRAM
	993 (0.74%)	636 (0.23%)	1,260 kbit (9.58%)

Table 5.10: Software frame rate for Sobel filter.

Platform	Frames/second
CPU (single core)	826.7
CPU (multicore)	594.1

Example output is given in 5.7.

Figure 5.7: After Sobel filter.

6
Discussion

This chapter contains reflections on our results. Suggestions for further development are also given, including discussion on what disparity map algorithm would be suitable for the platform evaluated or a similar one. Finally, the methodology and work process is briefly discussed.

6.1 Results and further development

The greyscaling is well optimized and works at speeds superior to the single core CPU when calculating eight pixels in parallel. This stage can be considered a clear success and well-suited for the platform.

It is clear from the results that the remapping will be a bottleneck for the system throughput. Remapping could have higher throughput if the spatial locality of rows would be used to process two or more output pixels from the same output row. Timing constraints of the project ruled out a serious attempt, but the mechanism has been implemented by Oh and Kim [36]. Furthermore Oh and Kim computed the remap transformation matrices online in real time to save off-chip memory bandwidth. But, in order to do that, division and square root operations have to be performed for each output pixel [36]. Using some data compression techniques and going for lower accuracy in the fractional part (10 or 8 bits instead of the 16 bits originally proposed) would be another way to lower the off-chip memory bandwidth of our current implementation. The necessity of such development depends on how the following disparity map stage would be implemented. Gribbon et al. propose to use a three point interpolation instead of fetching a single pixel in the up right fetch from Figure 4.9d [33]. This might be good if memory bandwidth to the input to CB is a limiting factor on throughput. Just using bigger port widths in our platform solves that issue reasonably well as can be seen from the port width testing in Section 5.2. However, the interpolation scheme proposed by Gribbon could improve the FPS by one or two frames.

The resizing currently is significantly slower than the software alternatives it has been compared with. There is significant room for improvement in the resizing stage primarily through more sophisticated memory usage - currently, the column-reduced frame is double buffered in order to simplify memory access patterns. The BRAM utilization would be lowered if the row-reduction worked horizontally along the image as rows became available, rather than working along columns when a whole frame is available. Speed could then also potentially be increased by using a wider port width. It was decided to leave this issue as is due to the time constraints

and other aspects that needed to be covered by the project, as well as the bottleneck of the remapping limiting throughput anyway.

In the current implementation, the Gaussian blur and Sobel filter, much like the resizing, are limited by memory port width. This is of little concern, given that they still work faster than the remapping and likely faster than any camera. It is likely that memory could be utilized more efficiently for both stages. Currently, full on-chip frame buffering is used, as the available memory allows it after the resizing. More importantly, it would be relatively simple to make the Sobel filter more accurate by implementing a fully pipelined square root circuit instead of the current approximation, as well as by normalizing the output rather than saturating it. The remapping limits throughput to such a degree that the double frame buffer between the filters could be changed to a single frame buffer in order to save memory.

As system throughput is limited to about 60 frame pairs per second by the remapping, there is no point in using the fastest implementations of other stages. The other parts of the system should rather use the slowest of the evaluated implementations in order to decrease power consumption.

The system could be ported to a cheaper platform. The most important change this modification would require is to move intermediate data into external memory. Using a System on Chip platform such as the Xilinx Zynq-7000 series is an attractive option, as the hard processor could handle input/output tasks, and perhaps run feature detection on-chip.

The stereo camera setup influences the system quite heavily as can be seen from Figure 3.7 in Section 3.6. Therefore it is important to set all the parameters correctly depending on the application details. For example, let's say a camera is set up in a ceiling that is 3.5 meters from the floor, and we want to detect humans in the cameras field of view. In that case the the camera has to be calibrated for having the horopter from 1 m to 3.5 m. That camera can then not be used if the ceiling is any higher but if it is lower than the horopter can be shrinked to 0.5 m or 0.25 m but that in turn will limit the area covered by the camera.

Our small study of disparity map calculations focused on block matching using sum of absolute differences and on belief propagation. The latter gives much better results, at the cost of increased complexity and higher required memory bandwidth.

In the case of block matching using SAD, the cost aggregation, similarity matching, and Winner Takes it All evaluation can all be performed in the same sweep across the frames. No intermediate values have to be stored, resulting in comparatively small memory requirements. Perri et al. [27] compare a few SAD implementations; most of the implementations could achieve a frame rate of more than 20 FPS with the frame size we use after resizing, with several of them using slower platforms than the one used here [27]. It would seem that block matching with SAD is very possible to implement on our platform or a similar one.

Tseng et al. made a block-based implementation of belief propagation on an FPGA platform [25]. Block-based belief propagation divides the frames into sections. In this particular case, the sections were 32 by 32 elements. With this approach, the required on-chip block RAM needed for processing one block is 172 KB. The part of the frames that are not in the block are stored in external memory since we need to keep the disparity space after each iteration of the algorithm. That puts a

great strain on the off-chip memory bandwidth and could result in the need of more expensive memory hardware structure.

6.2 Work process

Much less was achieved during the project than what was planned. The original intention was to write a hardware implementation for a chosen disparity map algorithm. About halfway into the project, this ambition was abandoned, as other development took too much time. A large amount of time was spent developing interfaces for on-chip testing, which in the end had to be scrapped as they were too time-consuming to develop. We had to move on in order to develop the image processing that the thesis was intended to cover. Final tests were done with simulations rather than running on chip. In what we have developed, we recognize several possibilities for improvement that might have been implemented if we had more time. If we were able to redo the project from scratch, we would probably have a smaller scope and use a platform with proper video input capabilities.

7
Conclusion

Out of the evaluated implementations, the slowest ones will be used for the greyscaling, the resizing and the filters, as total system throughput is limited by the remapping stage. The remapping stage will use a port width of eight pixels, as there seemed to be very little speed gain from increasing it further. With this configuration, about 60 frame pairs per second can be processed. Utilization and power consumption for the combined system with these choices is shown in Table 7.1. Utilization includes the DDR3 controller, but power estimations do not (due to simulation issues). The controller is expected to add a significant amount of power consumption, as it is responsible for about half the total logic utilization and runs at twice the clock speed as the rest of the system.

Table 7.1: Utilization and power consumption for the combined system.

LUT	23132 (17.23%)
FF	20004 (7.45%)
BRAM	9162 kbit (69.71%)
DSP	6 (0.81%)
Static power	0.145 W
Dynamic power	0.557 W
Total power	0.702 W

The software implementation performs the entire preprocessing chain at 84 FPS when using a single core and 128 FPS when using all available cores. However, according to Intel, the processor used dissipated an average of 47 W when performing demanding tasks, and the recommended shelf price is $570 [37]. For comparison, the FPGA used in the project costs $251.25 at one reseller [38], and the project could be scaled down to a cheaper platform. It is our belief that the camera will likely limit the speed in a real system, making the cost and power consumption a more important factor as long as the processing stays above the frame rate of the camera.

To summarize, it is clear to us from our results that an FPGA is viable for implementing a stereo video preprocessing pipeline. While the software implementation that we have compared with is faster, it was run on a expensive and power intensive desktop platform not suitable for embedded applications. Economical concerns may necessitate using a cheaper platform than what was evaluated in this project. The primary concern with this is the smaller amount of on-chip memory, which will likely require heavier use of external high-speed memory.

Due to the lower complexity and memory requirements, our suggestion for initial

development of a disparity map calculator would be to use a local method such as block matching using sum of absolute differences. Ways to refine and improve the method can then be pursued.

It might be advisable to perform further development on a System on Chip such as the Xilinx Zynq-7000 series in order to more easily integrate the processing in a complete camera system. The on-chip microprocessor of the Zynq platform could handle tasks outside the scope of the image pipeline itself, for example the IP stack in order to enable video over ethernet.

Bibliography

[1] J. M. Blackledge, *Digital Image Processing: Mathematical and Computational Methods*. Woodhead Publishing, 2006.

[2] Y. Wang and J. Kato, "Integrated pedestrian detection and localization using stereo cameras," in *Digital Signal Processing for In-Vehicle Systems and Safety*. Springer, 2012.

[3] N. Kehtarnavaz and M. Gamadia, *Real-Time Image and Video Processing: From Research to Reality*. Morgan & Claypool, 2006.

[4] M. Arias-Estrada and J. M. Xicotencatl, "Real-time FPGA-based architecture for stereo vision," in *Real-Time Imaging V, 59*, 2001.

[5] "Real-time image processing with dynamically reconfigurable architecture," *Real-Time Imaging*, vol. 9, no. 5, pp. 297 – 313, 2003.

[6] *AC701 Evaluation Board for the Artix-7 FPGA User Guide*. Xilinx, 2013.

[7] *7 Series FPGAs Overview*. Xilinx, 2014.

[8] *DDR3 SDRAM SODIMM*. Micron Technology, 2010.

[9] I. Point Grey Research. Bumblebee2 1394a. [Online]. Available: http://www.ptgrey.com/bumblebee2-firewire-stereo-vision-camera-systems

[10] *Vivado Design Suite User Guide: Implementation*. Xilinx, 2015.

[11] *Vivado Design Suite User Guide: Logic Simulation*. Xilinx, 2015.

[12] *Vivado Design Suite User Guide: Programming and Debugging*. Xilinx, 2015.

[13] *Vivado Design Suite User Guide: Designing with IP*. Xilinx, 2015.

[14] S. Dhanani and M. Parker, *Digital Video Processing for Engineers*. Elsevier, 2012.

[15] R. M. Haralick and L. G. Shapiro, "Glossary of computer vision terms," *Pattern Recognition*, vol. 24, no. 1, pp. 69 – 93, 1991.

[16] C. Georgoulas and I. Andreadis, "FPGA based disparity map computation with vergence control," 2010.

[17] C. Poynton, *Digital Video and HDTV*. Morgan Kaufmann, 2002.

[18] Z. Zhang, "A flexible new technique for camera calibration," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 11, pp. 1330–1334, Nov 2000.

[19] R. I. Hartley, "Theory and practice of projective rectification," *International Journal of Computer Vision*, vol. 35, no. 2, pp. 115–127, 11 1999, copyright - Kluwer Academic Publishers 1999; Last updated - 2012-10-21.

[20] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[21] D. Bailey, *Design for Embedded Image Processing on FPGAs*. Wiley, 2011. [Online]. Available: https://books.google.se/books?id=uvq7ImNssKwC

[22] P. Danielsson and O. Seger, "Generalized and separable sobel operators," in *Machine Vision for Three-Dimensional Scenes*. Academic Press, 1990.

[23] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47, no. 1-3, pp. 7–42, 2002.

[24] "Accurate hardware-based stereo vision," *Computer Vision and Image Understanding*, vol. 114, no. 11, pp. 1303 – 1316, 2010, special issue on Embedded Vision.

[25] Y.-C. Tseng, N.-C. Chang, and T.-S. Chang, "Block-based belief propagation with in-place message updating for stereo vision," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*. IEEE, 2008, pp. 918–921.

[26] S. Birchfield and C. Tomasi, "Depth discontinuities by pixel-to-pixel stereo," *International Journal of Computer Vision*, vol. 35, no. 3, pp. 269–293, 12 1999, copyright - Kluwer Academic Publishers 1999; Last updated - 2012-10-21.

[27] S. Perri, D. Colonna, P. Zicari, and P. Corsonello, "SAD-based stereo matching circuit for FPGAs," in *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, Dec 2006, pp. 846–849.

[28] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient belief propagation for early vision," *International journal of computer vision*, vol. 70, no. 1, pp. 41–54, 2006.

[29] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 23, no. 11, pp. 1222–1239, 2001.

[30] M. F. Tappen and W. T. Freeman, "Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters," in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*. IEEE, 2003, pp. 900–906.

[31] *7 Series FPGAs Memory Resources.* Xilinx, 2014.

[32] *Zynq-7000 AP SoC and 7 Series Devices Memory Interface Solutions v2.3.* Xilinx, 2014.

[33] K. T. Gribbon and D. G. Bailey, "A novel approach to real-time bilinear interpolation," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on.* IEEE, 2004, pp. 126–131.

[34] *MicroBlaze Processor Reference Guide.* Xilinx, 2014.

[35] J. Wiebe, private communication.

[36] S. Oh and G. Kim, "An architecture for on-the-fly correction of radial distortion using FPGA," pp. 68 110X–68 110X–9, 2008.

[37] (2015) Intel® core™ i7-4910mq processor (8m cache, up to 3.90 ghz). [Online]. Available: http://ark.intel.com/products/78939/Intel-Core-i7-4910MQ-Processor-8M-Cache-up-to-3_90-GHz

[38] (2015) Xc7a200t-2fbg676c xilinx inc | 122-1865-nd | digikey. [Online]. Available: http://www.digikey.com/product-search/en?mpart=XC7A200T-2FBG676C&vendor=122