FPGA Implementation of a Real Time Stereo Vision System

A dissertation
submitted in the partial fulfillment of
requirement for the degree of

MASTER OF TECHNOLOGY

in

Microelectronics and VLSI

Submitted by

Prathmesh Sawant
(173070039)

Under the supervision of

Prof. Sachin B. Patkar

Department of Electrical Engineering

Indian Institute of Technology Bombay
Powai, Mumbai - 400076

May 2019

Dissertation Stage-II Approval

The dissertation entitled

FPGA Implementation of a Real Time Stereo Vision System

by

Prathmesh Sawant

(Roll No. : 173070039)

is approved for the degree of

Master of Technology in Electrical Engineering

_____	_____
(Examiner)	(Examiner)
_____	_____
	Prof. Sachin B. Patkar
	Dept. of Electrical Engineering
(Chairperson)	(Supervisor)

Date:

Place: IIT Bombay

Acknowledgements

I express my gratitude to my guide Prof. Sachin Patkar for providing me the opportunity to work on this topic.

I would like to thank Mr. Mandar J. Datar for his guidance and ideas for my project work.

I would also like to thank Mr. Imran Syed, Mr. Rashish Shingi, Mr. Nikhar Gangrade for their suggestions and collaborative effort that went into developing the whole system.

I would also like to extend my deepest gratitude to my family for their support and encouragement.

Prathmesh Sawant

Abstract

Autonomous vehicles and drones require a 3D map of their environment so that they can function correctly and safely. The first step in 3D modelling is the estimation of depth. Depth estimation can be carried out using many active and passive off the shelf sensors operating on different principles. Some of the popular techniques are RGB-D cameras, Time of Flight (ToF) sensors and stereo camera systems. Stereo camera systems mimic the binocular vision in humans. They use images from two adjacent cameras to calculate the depth of a scene point by using epipolar geometry. Each of these methods have their strengths and drawbacks. IR based sensors do not perform well in outdoor environment due to heavy IR radiation from the Sun, They may fail when there are objects like glass which allow the IR rays to pass through or objects which absorb the radiation. The range of object detection depends on the power supplied to the IR blaster. Stereo Cameras work equally well outdoors and indoors but fail in low texture environments as they rely on features in the images to estimate depth. Increasing the range of detection in Stereo Camera systems is easy as it only involves increasing the distance between the two cameras. The major drawback in Stereo Cameras Systems is the compute intensive nature of the depth estimation algorithm. But these compute intensive algorithms have a high degree of parallelism which can be exploited on an FPGA.

The project objective is to implement a real time stereo depth estimation system on Zedboard (ARM-FPGA). Different camera sensors such as OV7670, ELP Camera, Intel D435i and Zed Camera are used for sensing images. Different window based stereo algorithms such as Sum of Absolute Differences (SAD), Census Transform and Semi Global Matching (SGM) are used for stereo depth computation. The system takes in real time data from the cameras and generates depth image from it. Rectification of the images as well as stereo matching will be implemented in the Zedboard FPGA. A VGA monitor is interfaced to Zedboard to display the computed depth image in real time. Also, data from a remote camera can be sent to Zedboard for processing using socket communication over the ethernet.

Contents

List of Figures

Chapter 1

Introduction

Stereo vision is based on mimicking the binocular vision of humans. In humans, each eye captures its own view and two such views are sent to the brain for processing. The two views have a lot of similarities but also have some differences. These differences are essentially shift in the position of objects relative to their eyes' field of view. This shift is larger for objects which are closer to the eye and almost nonexistent for far away objects. This shift is known as disparity in Computer Vision terminology.

Consider a setup in which two cameras are oriented in the same direction and separated by a constant distance along the horizontal axis. Let us define the image captured by the left camera as the reference image and the one captured by right camera as the target image. The right camera's field of view will be shifted slightly to the right as compared to the left cameras'. Thus the right camera image will contain some information on its rightmost parts which will be missing in the reference image. Also all objects in the right image will be shifted towards left as compared to the reference image. It can be proved [1] with simple geometry that the shift (disparity) will vary linearly with the inverse of distance of the object from the two cameras. The disparity can be found out by taking a pixel (x_l, y_l) in the reference image and traversing towards left to find its corresponding matching pixel (x_r, y_r) in the target image. This is known as the stereo correspondence problem. This step can be repeated for all pixels in the reference image, and an image from the disparity values for every pixel can be formed. The image will contain intensity levels which will correspond to depth of the object from the camera. Such an image is known as a disparity map or depth map. It may seem that for every pixel in the left image, one has to scan through all pixels in the right image which are towards the left of the reference pixel. However, using a process called Rectification, we can confine the disparity

along the row of the reference pixel.

Chapter 2 describes the relevant work done in the direction of the project. Chapter 3 familiarizes the reader with the implemented system. Chapter 4 describes the different tools and utilities used for development. Chapter 5 describes the data acquisition from cameras. Chapter 6 describes the SAD, Census and SGM Algorithms used for stereo matching. The disparity results on images captured from different cameras is also discussed here. Chapters 7 describes the observations and insights gained from the project. Chapter 8 describes the improvements or variations that can be implemented in the future.

Chapter 2

Related Work

2.1 Implementation of SAD Stereo matching Algorithm [1]

A stereo vision system was implemented on Zedboard[6] FPGA using Vivado HLS and SDSoC. Both of these are high level synthesis tools which allow us to write code in C/C++ which is later translated into HDL code. Vivado HLS was used for optimizing the algorithm for efficient implementation. SDSoC was used to implement the algorithm as hardware and software functions. Rectified input images from Middlebury University stereo image dataset [22] were stored in SD Card inserted into the Zedboard. The software functions ran on the ARM A9 processor on the Zedboard where as the compute intensive SAD stereo matching function was marked as hardware and was implemented in the FPGA. The communication between the ARM PS and the FPGA is managed by SDSoC. The software functions were responsible for moving the data back and forth between the SD card and the Zedboard DDR whereas the hardware function read input data from the DDR and stored the processed data into DDR.

2.2 Environment development for stereo vision with software-hardware interface [2]

The objective of this project was to develop a stereo based disparity system using software-hardware-co-design framework on Zedboard. The framework was able to retrieve left and right images from Android phone over the Ethernet, compute disparity in programming logic and send back the disparity image in a suitable format for user to view it. Robot Operating sys-

tem(ROS) library which was run on the Zedboard PS was used for viewing and communication. The framework also included data transfer between the PL and PS for real-time disparity computation.

2.3 Stereo Image Rectification Module implemented on FPGA [3]

Intrinsic and extrinsic calibration of individual stereo cameras is done to remove distortion effects. Once calibration is done, we perform rectification of stereo cameras. Rectification is the process of aligning the images generated from the two cameras such that the disparity is along the same epipolar line. This step is absolutely necessary for stereo matching to work. Camera calibration parameters are obtained by OpenCV C++ application program. The remap transformation requires data structures called maps which are generated using the obtained calibration parameters. Maps have the same number of elements as the image and each element is a pair of floats. Maps have to be generated once, but the transformation has to be applied on every frame. A remap module was implemented on FPGA which applies the transformation on raw images and generated rectified images in real time. From hereon we can assume that rectified images captured by cameras are available on which stereo matching algorithms can be applied.

2.4 Depth Map Estimation and Real-Time Implementation on FPGA [4]

Development of SGM stereo matching algorithm for FPGA was a collaborative effort with Mr Rashish Shingi.

2.5 Acceleration of Stereo Vision Algorithms using Jetson TK1 [5]

Stereo Vision algorithms were implemented on CPU and Jetson TK1 GPU. We use the CPU SGM implementation results for testing the correctness of our SGM FPGA implementation.

2.6 AXI VGA peripheral

The AXI VGA peripheral displays data stored in the DRAM of FPGA onto a VGA monitor. This is used to display the disparity image in real time. This was developed by Mandar J Datar, PhD student HPC lab, using his own Python to HDL converter.

Chapter 3

Overview

The FPGA board used for development is Zedboard[6]. Figure 3.1 shows the overview of the implemented system. Left and right images captured from the cameras are stored into DDR RAM (off chip RAM). Maps required for the remap transform are generated offline and stored into DDR RAM. We need two Remap peripherals for the left and right images respectively. The Remap peripheral reads the raw frame and the corresponding map and generates rectified frame. The rectified images are again stored into DDR. The stereo matching peripheral (SGM block in figure) then reads the left and right rectified frame and generates disparity image which is again stored into DDR. The AXI VGA peripheral is configured to display the disparity image onto a VGA monitor.

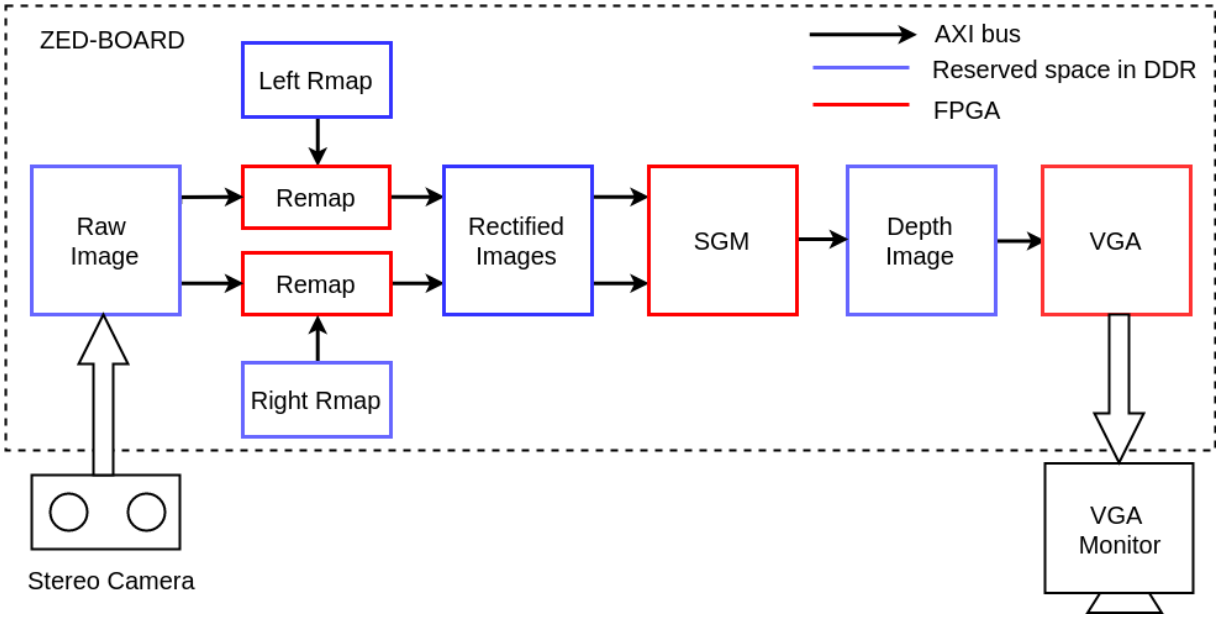


Figure 3.1: Block diagram

Zedboard is an SoC which has a hard ARM processor and Zynq 7020 FPGA chip. Capturing data from USB cameras and controlling the different peripherals is done via application programs which run on the ARM processor. The peripherals which do the computations are implemented in FPGA.

The resolution of images is fixed to 640x480 and cameras are configured accordingly. Each pixel is stored as an eight bit number. The metric used to profile the computation times of different peripherals and also the cameras is fps (frames per second). From here on a frame means 640x480 pixels.

We could have skipped storing the rectified images and passed the output of the Remap peripheral directly to the stereo matching peripheral. We chose not to do this because our performance is not limited by memory read-write but by the FPGA peripherals themselves. We use AXI4 protocol to perform memory read-write. Zynq 7020 chip[7] supports 4 high performance slaves which can be used to access DDR RAM. Each slave can stream 64 bit data per clock cycle. Let us assume that 30% of the clock cycles are used for data request and handshaking and data is transferred on remaining 70% clock cycles. We operate the Zedboard on 100MHz. By calculation, we see that we can read or write at 7291 fps which is orders of magnitude greater than the FPGA peripherals.

Chapter 4

Workflow

Figure 4.1 shows the workflow of the project. The blocks represent the various tools used for development. The tools are discussed in the subsequent sections.

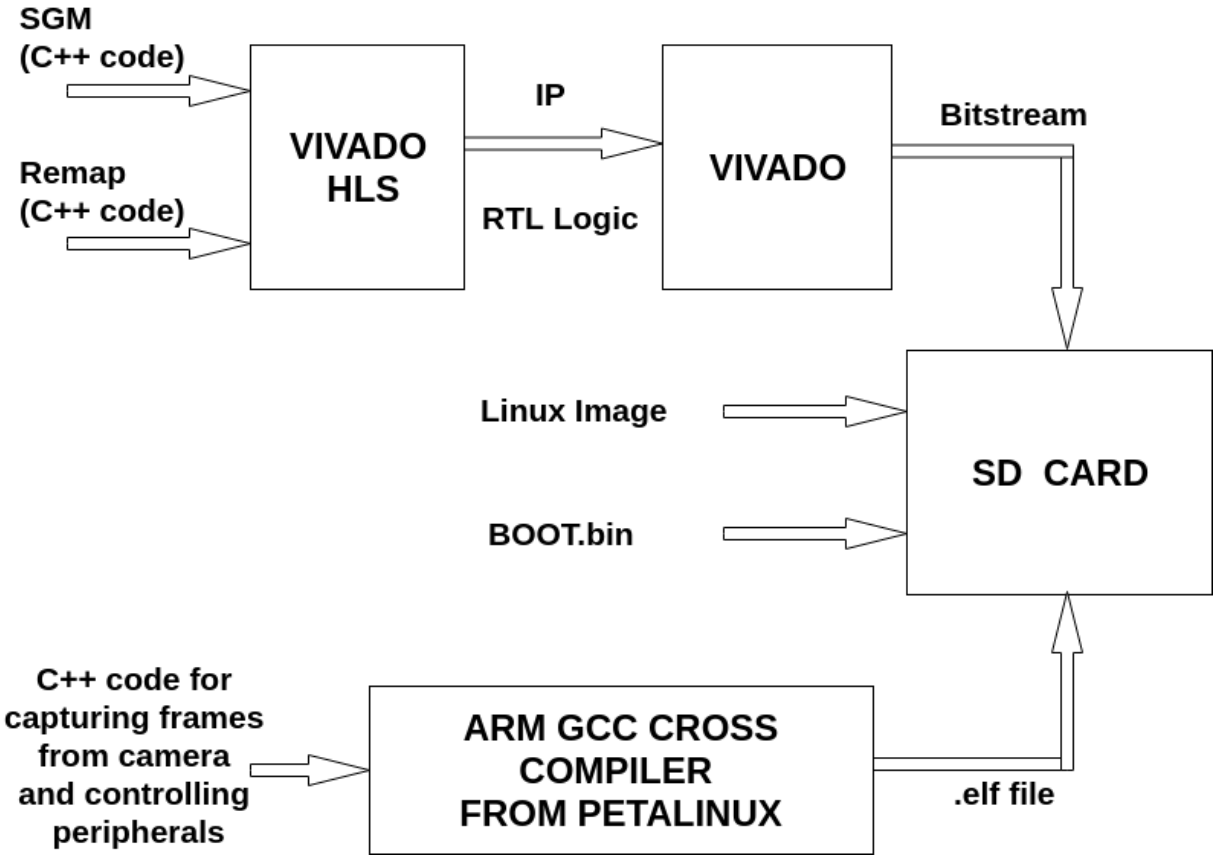


Figure 4.1: Workflow of the project

4.1 Vivado HLS 2017.1[8]

HLS stands for High Level Synthesis. Vivado HLS allows us to write logic in C/C++ which is then transformed to Hardware Descriptive Language (HDL) (VHDL, Verilog and System C). This allows rapid prototyping of our design and also drastically cuts down the time required to implement complex stereo algorithms. The same C/C++ code can be tested using any C compiler (eg gcc/g++). For testing purposes, a C/C++ testbench is required which calls the top level function in HLS.

Certain guidelines have to be followed while writing HLS code. One has to think about what hardware will be inferred from the lines of code. For eg. division or mod operation by any number other than powers of 2 will be inferred as a complicated logic; 'for' loops which 'break' at a particular iteration will be inferred as complicated exit checks which happen at every iteration. Also recursive functions cannot be synthesized into logic. It has been observed that writing C code in RTL like fashion in which data is read from a register, processed and then stored into another register get synthesize efficiently.

Once the correctness of the code is verified using a testbench, one can synthesize the code. This generates the corresponding HDL. The tool also provides an estimate of resources consumed in the FPGA, required clock period, and the number of clock cycles required. It is important to note that these are just estimates and many-a-times the tool gives a pessimistic estimate of resources sometimes exceeding 100% but it is later observed that the design fits in the board. The clock period estimate is also pessimistic and one can safely ignore clock period slightly greater than the target clock. The estimate for number of clock cycles required is quite accurate which is good because it is a measure of computation time required by the peripheral.

Pragmas are guidelines to the tool for synthesizing logic. They are inserted within the code. Pragmas have a variety of uses like specifying the interface for the peripheral, for unrolling a loop in the implementation. If we observe (from the estimate reports generated by the tool) that the implemented logic is slow or takes up a lot of resources, we can change certain pragmas or we can rewrite certain parts of the code till satisfactory results are obtained. One can go the 'analysis perspective' in the tool to observe the computation cycles required for different operations in the code. Once satisfactory results are obtained, one can do a C/RTL Cosimulation which simulates the RTL generated using the C test bench. The pragmas used are ignored by standard C compilers. Some of the pragmas are discussed in subsequent chapters in this report.

We may want to have arbitrary number of bits in our logic and not be limited by data types like short, int, long etc. For that reason we have ap_int.h library. We can define signed or unsigned data types with arbitrary number of bits. Also any bit of a scalar element can be accessed using C like array indexing. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this ap_[u]int. The ap_int.h file is located in the directory HLS_ROOT/include, where HLS_ROOT is the HLS installation directory. Details about ap_int.h are available in [8].

Vivado HLS provides us the option to package the generated HDL into an Intellectual Property (IP) block. These IP blocks can be used in Vivado projects. A TCL script can be run instead of using the GUI for repeated faster use.

4.2 Vivado 2017.1[9]

Vivado generates bitstream from HDL source files or IPs. The bitstream is then used to program the FPGA. We can either use HDL source files to synthesize our design or use the 'create block diagram' option to add source files or IPs graphically. We use the graphical method to add our HLS IPs and source files to the design.

4.3 AXI Interface

An AXI IP has master and slave interface(s). The AXI master does the job of reading/writing into memory. The AXI slave interface has certain registers. To explain the use of these registers we will consider our own stereo peripheral. Let's say our peripheral reads left image from address L, right image from address R and writes disparity image into address D. Our AXI IP will have the following AXI slave registers- Control Register: to start the peripheral and observe the status of the peripheral, Input Left Register: Here we are supposed to write the address of (first pixel of the) left image (Address L) stored in DDR. This is an indicator to the master that it has to start reading the left image from this location, Input Right Register: similar register for the right image, Output Disparity Register: Here we are supposed to move the address where we want the disparity image to be stored (Address D). The AXI registers are just some address locations in DDR. For eg. If the left image is stored in DDR at location 0x10000000 and the Input left register is located at 0x43C00004, then we have to move data 0x10000000

into address 0x43C00004.

4.4 Software Development Kit 2017.1 (SDK)

SDK can be used to compile C/C++ programs for the hard ARM processor on Zedboard. The executable generated by SDK can be run on the ARM processor. We can select the platform to be bare-metal (No operating system) or linux. We mainly use these programs to control the FPGA peripherals and capture frames from the camera. Details are provided in next chapters. Although we can use SDK to cross-compile our C/C++ programs for the ARM processor, we prefer doing that with the arm gnu cross compiler which comes with Petalinux. SDK is still the only way to cross-compile C/C++ programs for bare-metal application.

Another, more important use of SDK is to be able to view the system.hdf file which contains the addresses of all axi slave registers (mentioned in the previous section).

4.5 XSCT [10]

XSCT is a command line tool which enables us to access the DDR RAM of Zedboard from a computer. XSCT can only be used for a baremetal application. As mentioned in the previous sections we need to move certain values into certain registers which are located in DDR RAM. One way of doing is to write a C program and cross-compile it for arm using SDK. Another way of doing it is through memory read/write commands in XSCT. XSCT also provides commands to dump contents (bytes) from a file to DDR and vice-versa. This is helpful while testing as we can write left and write images to DDR and read the generated Depth images without the requirement of a camera.

One can also write a TCL script which can be run on XSCT which will program the FPGA, run the elf (generated from SDK) for C program and do memory read/write operations.

4.6 Linux XSCT

A tool like XSCT is not available for Petalinux. But considering the usefulness of XSCT, a C based XSCT like interface was developed by Mr Mandar J. Datar. The C code is crosscompiled using arm gcc crosscompiler and the generated elf is run on the ARM processor. It supports

memory read/write, memory read into file and memory write from file. It is used for controlling the FPGA peripherals when we use a Petalinux environment.

4.7 Petalinux 2017.1

Petalinux is an open source Operating System(OS) by Xilinx which can be booted on the hard ARM processor on Zedboard [6]. It is easier to use USB cameras with an OS installed over the processor. Also having an OS enables internet connectivity which is required for communicating with a computer. When we boot the board with Petalinux, we have to also pass the bitstream. The ARM processor on Zedboard is booted with Petalinux and the FPGA is programmed with the provided bitstream. As we discussed in Chapter 3 the FPGA peripherals read data from DDR and write processed data to DDR. The same DDR is used by Petalinux to allocate memory for different processes. Hence, we have to reserve some memory which should not be used by Petalinux. This is done by modifying the device tree file while generating the OS image. Petalinux provides tools for cross-compiling C/C++ files for the ARM processor. The generated elf files can be run on the ARM processor.

4.8 Python

XSCT or Linux XSCT allows us to dump contents of a file to DDR or viceversa. The files bytes are dumped. So we have to convert our left and right images into a binary format so that they contain only pixel data as successive bytes. Also we have to convert the depth image read (bytes) into a suitable format for viewing. This was done using python scripts.

Chapter 5

Camera Interfacing and data capture

5.1 Custom Camera Rig

A custom stereo rig was made using two OV7670 [11] cameras. The OV7670 is a low cost, low voltage CMOS image sensor with a VGA resolution and inbuilt image processor. It is capable of operating at 30fps for VGA resolution. The cameras were connected to the FPGA PMODS and could be accessed directly from the FPGA.

The cameras are operated in their default configuration in which 4 bytes of data represent two colour pixels. The color model is YCbCr. The color channels Cb and Cr are shared between two adjacent pixels. The intensity channels are different. Out of 4 bytes, the first byte represents the shared Cb, second byte represents the intensity of Pixel 1, third byte represents the shared Cr and fourth byte represents the intensity of Pixel 2. For stereo matching, we do not need color information. Hence we can read only the even numbered bytes and ignore the bytes containing color information.

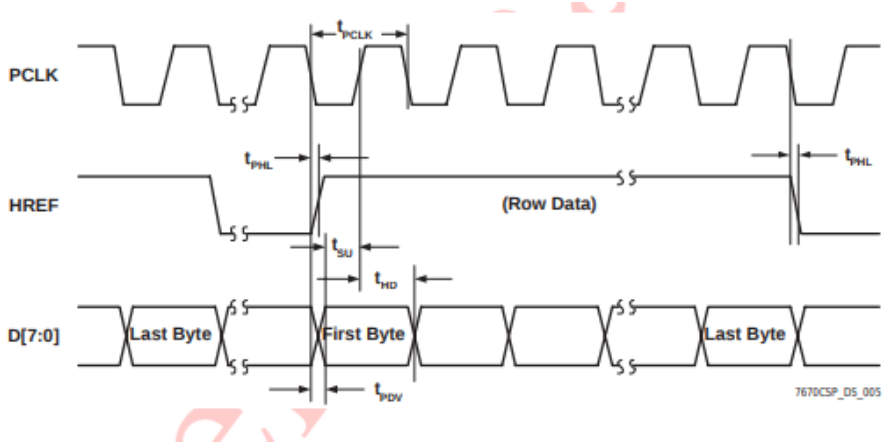


Figure 5.1: OV7670: Horizontal timing[11]

The timing signals provided by the camera are vsync and href. A high to low transition on vsync indicates the beginning of a new frame. The period for which href is held high is the row width. Only the data which arrives when href is asserted high is considered valid. The timing diagrams for href and vsync are shown in Fig 5.1 and 5.2. The relative distance between the

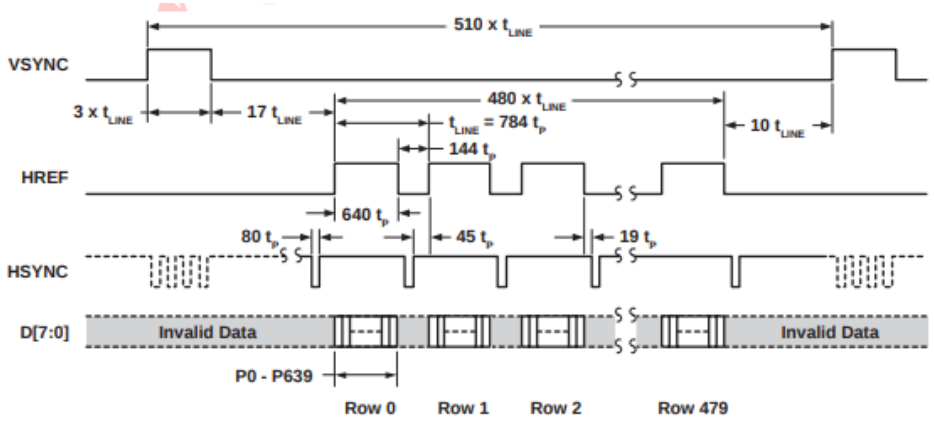


Figure 5.2: OV7670: VGA timing[11]

two cameras needed to be fixed so that they could generate a pair of stereo images. Also the two cameras should be at the same elevation to constrain the disparity along one axis. The distance between the two cameras is an important parameter as it decides the range of depth detection for the setup. Beyond this range, all objects will appear equidistant.

Our objective was to build a mount for both the cameras such that the distance between them was constant during operation but could be varied as per our requirement otherwise. We designed a PCB[1] mount for each camera. Each PCB mount had connectors over which two such mounts can be joined together. Fig. 5.3 shows the PCB setup. Connector strips are

[1]Design of the PCB was done by Nikhar Gangrade, M.Tech student HPeC lab

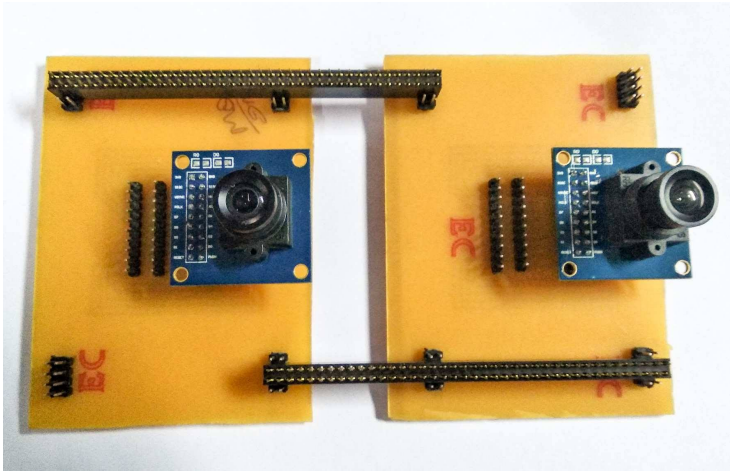


Figure 5.3: PCB mount

used to join the two PCB mounts. The distance between two mounts can be changed within a reasonable range as per our requirement. Although this setup provided the required stability, routing the signals through the PCB created noise and timing problems. One of the camera could not function properly when connected through the PCB. To overcome this we made a different mount as shown in Fig. 5.4 The cameras were mounted on a single acrylic sheet and wires are connected to the cameras directly. Using this mount we were able to stabilize both

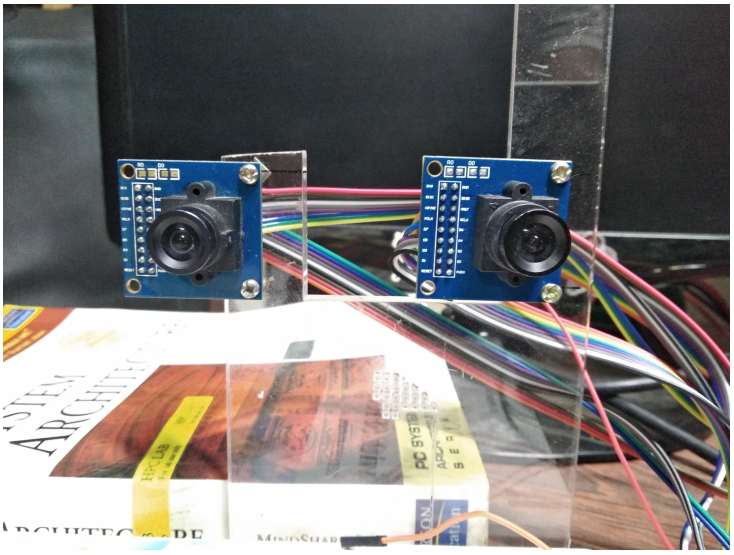
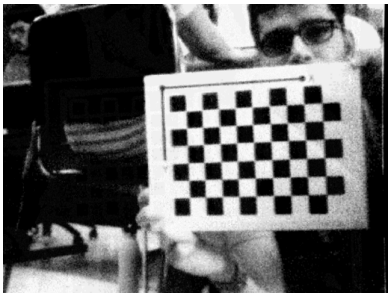
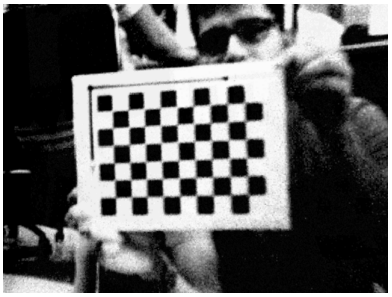


Figure 5.4: Acrylic mount

the cameras. The disadvantage with this mount is that the distance between the cameras is now fixed.



(a) Rectified left camera image



(b) Rectified right camera image

(c) Disparity image with intensity scaled

Figure 5.5: Images captured from OV7670 camera

Images captured from the cameras were stored into DDR memory. It was observed that the captured images contained a lot of noise. The results of stereo matching even after rectification were not satisfactory. Such a custom mount is also prone to relative movement between the cameras which will require re-caliberation of the cameras. Hence, it is much easier and safer to use an off the shelf stereo camera module to capture images. Figure 5.5 shows the captured and rectified images and the generated depth image.

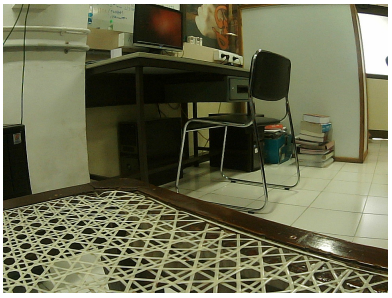
5.2 ELP Camera [12]

The ELP (model no. ELP-1MP2CAM001) camera is a USB Universal Video Class (UVC) type camera and can be accessed by the processor side of Zedboard. An application program (C++) was written to capture data from the ELP camera in the required format and store the captured frames in the reserved memory location. For eg. Camera 0 image is stored at address A, Camera 1 image stored at address B. Now, we start the stereo matching peripheral which reads from addresses A and B and stores the disparity image at address C. This application program will run on the ARM processor on Zedboard

The application program uses v4l2 library to capture images. We have also compiled

OpenCV for Petalinux, but v4l2 is preferred because it is lightweight which will result in faster operation. The ELP camera is detected as two different UVC complaint devices: /dev/video0 and /dev/video1. We tried multi-threading to capture the images from both cameras simultaneously which worked fine when we were capturing and writing compressed images to a file. But when we started storing uncompressed images (which is a must for stereo matching) (size 640x480), it complained that there was insufficient USB bandwidth available. However it could capture 320x240 resolution images from both the cameras simultaneously. This is a limitation of USB 2.x. The same problem has been observed while interfacing and simultaneously capturing images from multiple webcams connected on USB 2.x ports on a computer. The solution to this problem is to use USB 3.0 which provides higher data rate. A workaround to this problem is to capture compressed images and store them to files. Then read those images again and store the uncompressed images to DDR. Both of these operations could be done parallely for both the cameras.

The ELP camera supports a maximum resolution of 920x760. However there is visible radial distortion (fish eye effect) in the image as shown in Figure 5.6. This problem can be solved partially using rectification. We crop the central 640x480 section from the 920x760 image as it contains the least radial distortion so that rectifying the image is easier. Indoor and outdoor images have been captured using the ELP camera. Some of the images are shown in Figure 5.7



(a) 920x760 image with radial distortion

(b) cropped 640x480 image

Figure 5.6: Images captured from ELP camera

(a) ELP chair left image

(b) ELP chair right image

(c) ELP classroom left image

(d) ELP classroom right image

(e) ELP stairs left image

(f) ELP stairs right image

(g) ELP cycle stand left image

(h) ELP cycle stand right image

Figure 5.7: Sample images from dataset captured with ELP camera

5.3 Intel RealSense D435i [13]

Images from D435i camera can be captured only with the RealSense SDK functions. The RealSense SDK has been made opensource by Intel. USB 3.x interface is a required to capture image streams. Due to this limitation it could not be interfaced to Zedboard directly. So we

captured the left and right infrared images on a computer using a C program. These images can be sent to a socket client on Zedboard where they can be processed and disparity image can be generated. The disparity image can be sent back to the server (computer) for viewing.

The camera supports maximum resolution of 1280x720. There are 4 video streams which can be captured- depth, infrared left, infrared right and color. We capture the left and right infrared streams. It is observed that unlike traditional cameras, the left and right cameras do not have an infrared cutoff filter. Due to this sources of illumination or well illuminated objects especially in sunlit environments appear over exposed. We might need to preprocess these images before performing stereo matching. Figure 5.8 shows some of the images captured using the RealSense camera.

(a) RealSense stairs left image

(b) RealSense stairs right image

(c) RealSense foyer left image

(d) RealSense foyer right image

(e) Realsense cycle stand left image

(f) Realsense cycle stand right image

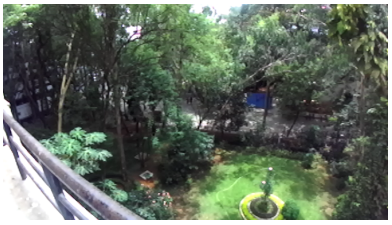
(g) RealSense classroom left image

(h) RealSense classroom right image

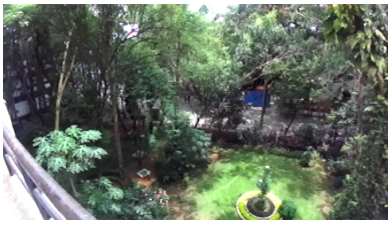
Figure 5.8: Sample images from dataset captured with Intel Real Sense camera

5.4 Zed Stereo Camera [15]

ZED stereo camera is UVC compatible. So data can be read using standard V4L2 library over USB interface. However using USB 2.0 only WVGA resolution images can be transferred since the data transfer speed is limited. In USB3.0, we can read full resolution HD images. The Zed Camera is detected as a single UVC complaint device. The frame obtained from the camera has the left and right images concatenated side by side. These images are captured using a C progem running on the arm processor and the images are appropriately saved in the DRAM for processing in FPGA.



(a) aerial left image



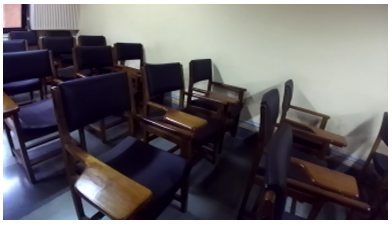
(b) aerial right image

(c) lab left image

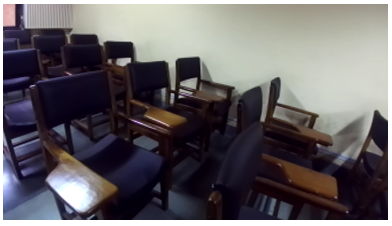
(d) lab right image

(e) outdoor left image

(f) outdoor right image



(g) classroom left image



(h) classroom right image

Figure 5.9: Sample images from dataset captured with ZED stereo camera

Chapter 6

Stereo Matching Algorithms

Stereo correspondence matching is the process of finding matching pixels in two images of a scene taken at the same time instant but from slightly different viewpoints and then transforming 2D positions to 3D profiles [16]. The shift between the positions of two matching pixels is known as disparity. The shift is measured for a target image (generally right image) with respect to a reference image (left image). Disparity is linearly proportional to the inverse of distance of the object from the base-line (distance between the two cameras). Disparity can be localized to a single axis by a process called Rectification. We have an FPGA implementation of image rectification module[3] which stores rectified images into DDR RAM. Hence we can assume that rectified images from cameras are available.

Stereo matching algorithms can be pixel based or window based. In pixel based algorithms at each pixel (x_i, y_i), the disparity is evaluated by finding the intensity differences, and the pixel position with the minimum difference value is considered for disparity at that pixel position (x_i, y_i). The difference can be absolute difference or square of absolute differences. Pixel based matching is shown in Fig 6.1. One important parameter is search range, i.e the number of pixels which are matched with a given pixel. If the disparity for an object is larger than the search range, then its depth will not be correctly estimated. Increase in the search range results in a linear increase in the number of computations.

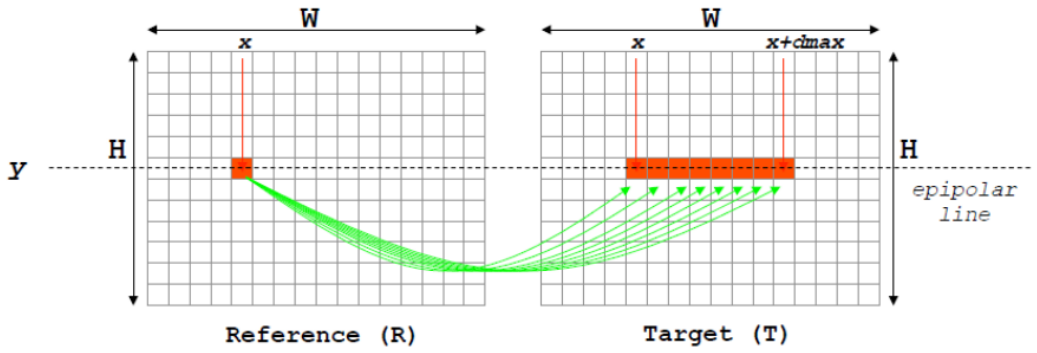


Figure 6.1: Pixel based matching [24]

Pixel based algorithms can give erroneous results as they compare only single pixel values. Window based algorithms make a window around the reference pixel. Matching cost computation is done window wise i.e the reference window is compared to windows around the pixels in the target image for a defined search range. Increasing the window size causes more of the image features to be compared but increases the computational cost in square of the window size. Window based stereo matching is shown in Fig. 6.2.

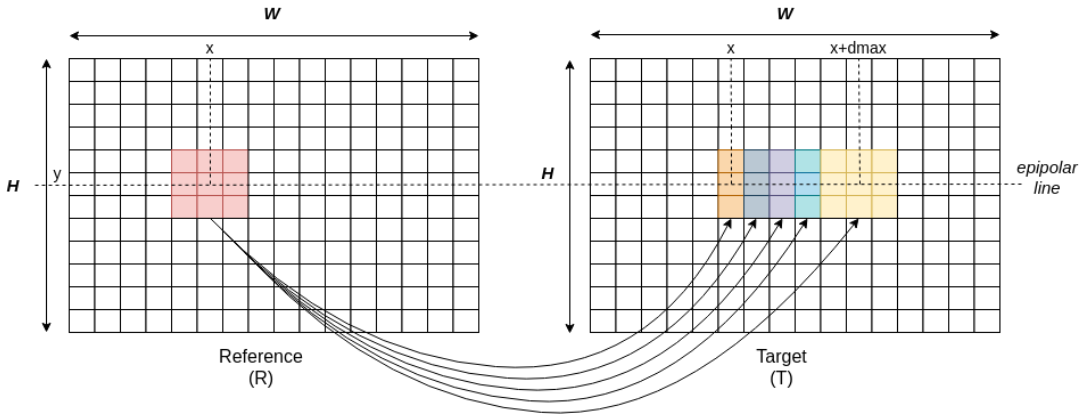


Figure 6.2: Window based matching

Three stereo matching algorithms have been implemented on Zedboard FPGA. They are Census Transform [18], Sum of Absolute Difference [19] and Semi Global Matching [20]. All of these implementations were done using High Level Synthesis (HLS) where C/C++ code is used to generate RTL logic and finally the bitstream used to program the FPGA. The C/C++ code when run on software generates the exact same output as the hardware implementation. The input image size is considered as 640x480 for all implementations. The input images are stored in DDR RAM. The stereo matching core reads the stereo pair computes depth image which is again stored into DDR RAM. The frame rate mentioned in the implementations in-

cludes the time required for reading images from memory, computing the depth image and storing the depth image back into memory.

The implemented stereo algorithms are discussed in the subsequent sections.

6.1 Census Transform [18]

Census Transform is a window based stereo matching algorithm. The windows are first transformed by comparing every element of the window with the central element. If the element is larger than the central element, we assign it as 1 otherwise we assign it as 0. Thus we form a vector of booleans 'census vector' for the left pixel as well as the right pixel. Then we compute the Hamming distance between the two vectors. The Census cost is the sum of bits of the Hamming distance vector. The index of the minimum cost over 'search range' is the disparity.

The window size is 11x11 and the search range is 80. Two different implementations of Census Transform were realized. They are discussed in the following sections

6.1.1 Implementation 1

Vivado HLS has many useful pragmas[17] which can be used to guide the tool to generate a suitable implementation. For understanding the implementation two important loop pragmas will be briefly discussed here.

- Loop Unrolling

  The UNROLL pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor N, to create N copies of the loop body and reduce the loop iterations accordingly. When a loop is unrolled into N iterations, the resources required for the body of the loop are replicated N times.

- Loop Pipelining

  The PIPELINE pragma allows the operations within a loop to be launched in a pipelined manner. For eg suppose there are 3 operations within a loop: A, B and C. The hardware inferred for each of these operations is unique and takes up one clock cycle for performing the operation. The loop runs for 2 iterations. In the case where the loop is not pipelined,

the operations are performed in this sequence: 1A, 1B, 1C, 2A, 2B, 2C. The time required to execute the loop is 6 clock cycles. When the loop is pipelined we can launch 2A along with 1B. Similarly 2B can be launched with 1C. The sequence of operations then is: 1A, (1B 2A), (1C 2B), 2C. It requires 4 clock cycles to execute the loop. The increase in resources due to loop pipelining is negligible.

A limitation of the tool is that if a loop is pipelined, all the loops inside that loop are fully unrolled.

The stereo matching HLS code has the following structure:

```
void stereo_matching_function ( )
{
    row_loop:for(int row=0; row<IMG_HEIGHT; row++)
    {
        col_loop:for(int col=0; col<IMG_HEIGHT; col++)
        {
            sr_loop:for(int d=0; d<SEARCH_RANGE; d++)
            {
                //match left_window with right_window[d]
            }
        }
    }
}
```

We iterate over 'search range' number of pixels in the right image (target image) for each pixel in the left image (reference image). This is repeated for all the pixels in the left image. The HLS tool was directed to pipeline the inner most loop i.e sr_loop (iterating over search range number of pixels) and reported a pipeline depth of 88 clock cycles. There are no operations between the row and col loop, hence they can be effectively flattened into a single loop. The time required for processing one frame for such an implementation can be given as

$$T \propto no.\,of\ rows \times no.\,of\ columns \times (search\ range + pipeline\ depth)$$

The resources available in Zedboard are as follows- 280 BRAM_18K(Block RAM), 220 DSP48E(Digital signal processors), 106400 FF(Flip FLops) and 53200 LUT(Lookup Tables). An initial implementation of Census had the following resource estimates in Zedboard: BRAM:8% DSP:0% FF:4% LUT:12%.

The logic synthesized roughly corresponds to matching of two Census windows along with registers to store data for the next iterations. As we sequentially iterate over rows, columns and

search range we reuse the same hardware. Thus, the FPGA resources required are independent of number of rows, columns and search range but computation time required is proportional to these parameters as shown by the above equation.

This gives us the idea that we can divide the images into a number of sections and process the sections independently. We can do this division along the image rows, columns or along the search range (search range being divided into sub ranges processed parallely). It is also apparent that as the window movement and search range increment is along the columns, the least expensive way in terms of extra logic required would be splitting the images along the rows.

We use multiple Census IP blocks to process these sections independently in parallel. We are limited by the most used resource in the implementation. In our implementation LUTs are the most used resource (12% of total). Hence we can split the image into at the most 8 sections (12*8 = 96 < 100) and use 8 Census IP blocks for processing. The frame rate thus becomes 8 times the original frame rate. The resultant frame rate obtained is 22 fps. We have demonstrated that our implementation is scalable. Given more FPGA resources, we can divide the image into more sections and have that many Census IP blocks processing in parallel, thereby increasing the frame rate.

The depth image contains some salt and pepper noise which is removed using a median filter. A mode filter was also implemented to remove the noise. But median filter performed better than the mode filter and also consumed less resources. The median filter is a square window based filter. To optimize the median filter, we used an approximation which is as follows. We first take median of values along a window row. This is done for all the window rows. Thus each row is reduced into a single element. Then we take the median of all such elements. The approximation we have used is that median of median of subgroups of data is the true median. This helped us halve the LUT consumption while keeping the filtered image quality intact. The extra logic required for the filter is comparable to a Census IP block. Also the filter is fast enough so that the frame rate is unaffected.

A note about sectioning the images. For any window based stereo algorithm, the upper and lower window_size/2 of the depth image has garbage values. This is shown in Figure 6.3 a. Now suppose we divide the input image pair into two sections and process them separately and then place the depth image generated one below the other, we will get a depth image of the entire scene with garbage values in the center. This is shown in Figure 6.3 b. This effect is

also shown in Figure 6.4 in which the input image pair was divided into 8 sections. Instead of random values, we have zeros because the DDR memory was written with zeros before starting the stereo peripheral. A solution to this problem is while sectioning the images, append window_size/2 number of rows to the top and bottom of each section. Thus height of each section is image_height/N + window_size/2 + window_size/2. While writing the disparity image, we write only the valid pixels. This is shown in Figure 6.3 c.

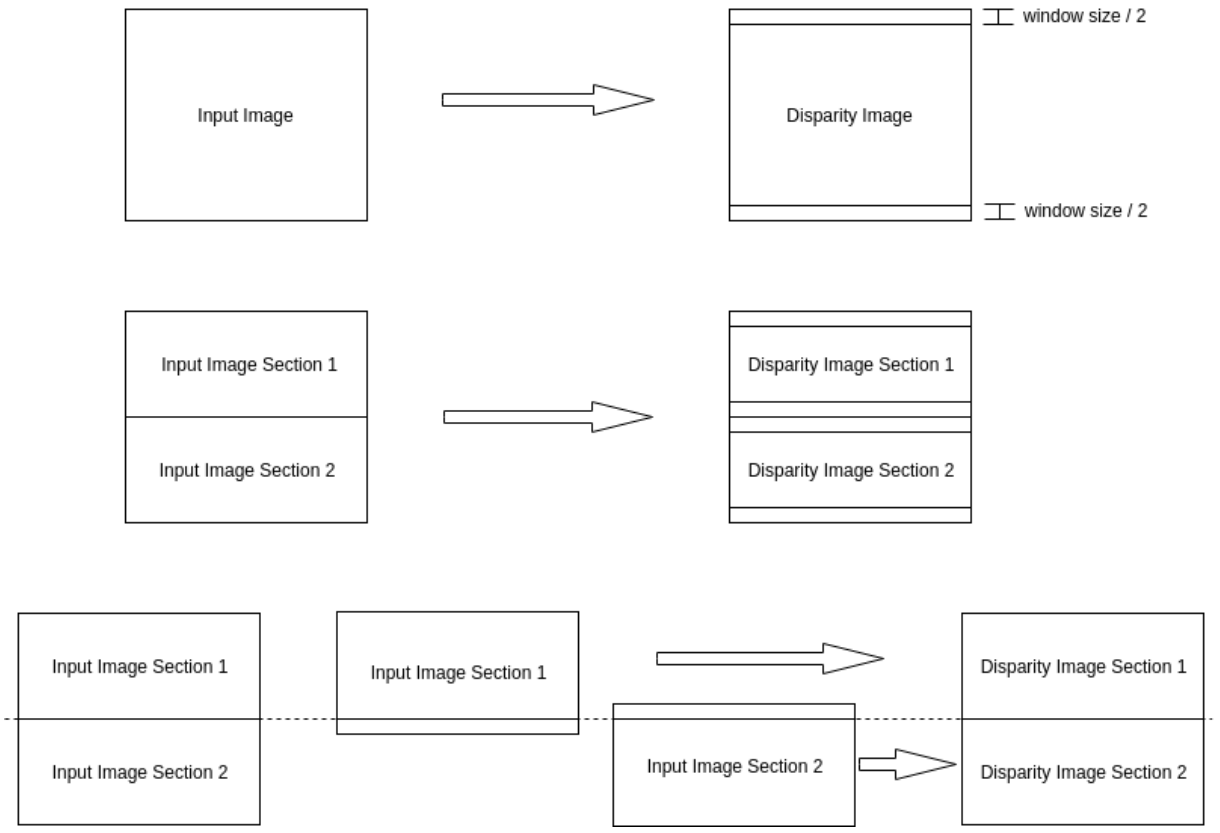


Figure 6.3: Dividing the input image into two sections to be processed by two blocks simultaneously

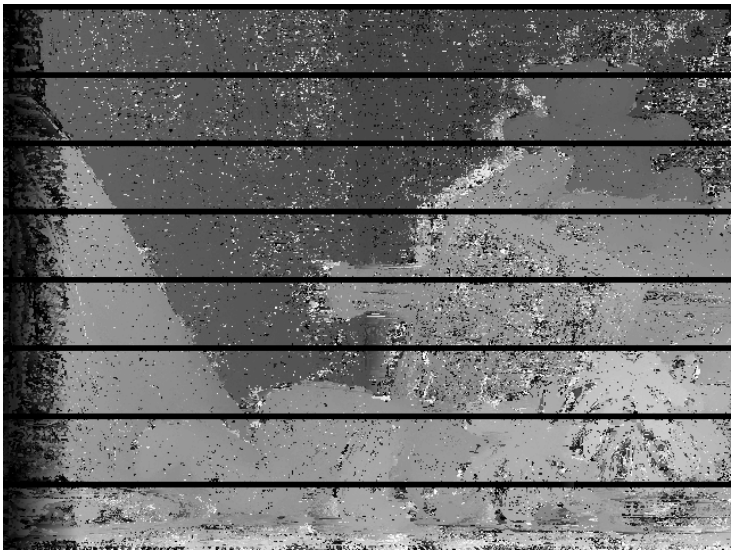


Figure 6.4: Census Disparity image generated using 8 Census blocks processing 8 sections in the image

6.1.2 Implementation 2

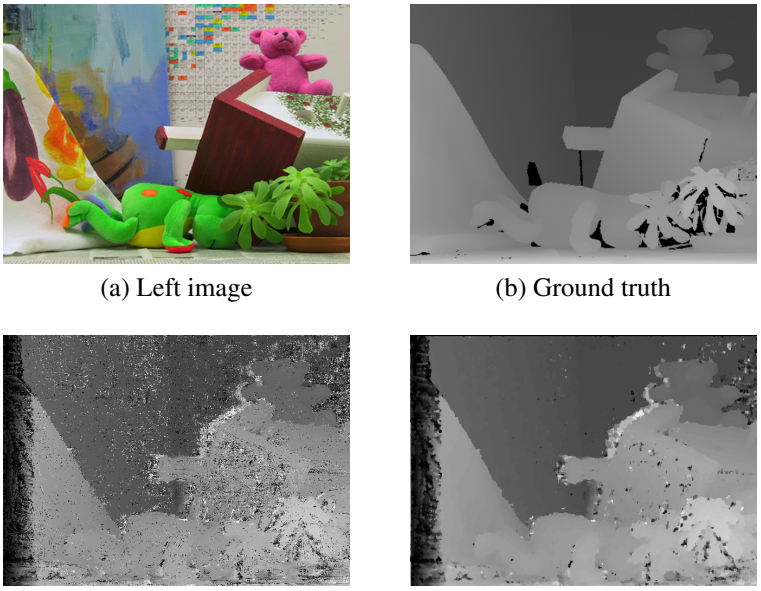
In Implementation 1 we had first tried to pipeline the merged row_col loop. This meant that the inner sr_loop had to be fully unrolled. But the onboard resources were not enough, hence it could not be implemented. To unroll the loop, we have to either decrease the search range which will degrade the quality of depth image obtained or we have to reduce the operations in the sr_loop. The operations inside the sr_loop can be divided into two groups-

- Obtain census vectors for a given left or right pixel.
- Compute disparity by comparing the array of right Census vectors with left census vector.

These operations are fairly independent. Taking advantage of this, we implemented Census stereo matching as a two stage process. The first stage peripheral reads pixels from the input image and stores Census vectors corresponding to each pixel into DDR. We use two instances of this peripheral for the left and right image respectively. The second stage peripheral, reads census vectors for the left and right image and computes disparity using Hamming distance between the vectors. As the operations inside the sr_loop were reduced, we could unroll the sr_loop in both the peripherals. The entire design could fit on Zedboard with a LUT (most contented resource) utilization of 61%. The first peripheral works at a frame rate of 332fps. The second peripheral works at a frame rate of 166 fps. As they work in a pipelined fashion, the over all frame rate is 166 fps.

An image corresponds to 640 x 480 x 8 bits or 307200 Bytes. However each Census Vector is a 121 (11x11) bit number which is padded to 128 bits. Thus storing census vectors corresponding to an entire image is like storing 64 images. But it is not a major concern as we have 512 MB DDR RAM at our disposal. Also the memory access times are much less as compared to the computation time of peripherals. But this implementation can be optimized by connecting the first stage and second stage peripherals using a FIFO.

Figure 6.5 shows the results of census stereo matching on images from Middlebury dataset. [22]



(a) Left image (b) Ground truth

(c) Census depth image (d) Filtered Census depth image

Figure 6.5: Census results on Middlebury images

6.2 Sum of Absolute Differences (SAD) [19]

Sum of Absolute Differences (SAD) performs the absolute difference between each pixel intensity in the reference image window and the corresponding pixel in the target image window. The aggregated difference value is calculated for a window. The index of the window where the aggregated difference value is minimum is taken as the disparity.

$$SAD\ Matching\ Cost = \sum_{i=-N}^{i=N} \sum_{j=-N}^{j=N} | I_L(i+x_0, j+y_0) - I_R(i+x_0+d, j+y_0) |$$

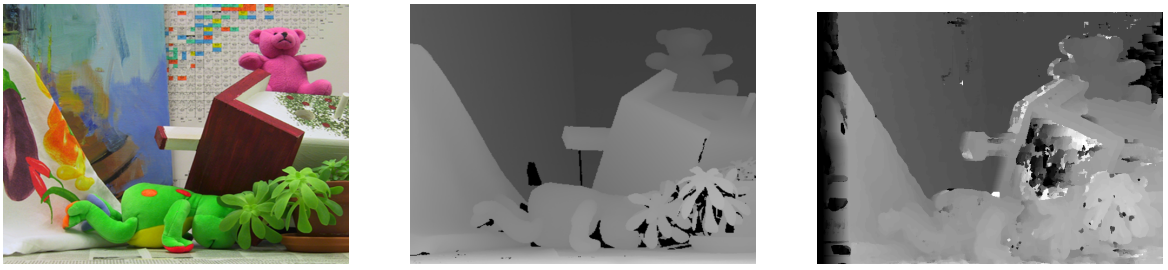
$(2N+1)(2N+1)$ is the window size. I_L and I_R are intensities of pixels in left and right images.

The implementation details are as follows. The resource estimates reported by HLS tool for SAD implementation were as follows- BRAM: (Block RAM):8% DSP(Digital signal processors):0% FF(Flip FLops):6% LUT(Lookup Tables):22%. We followed the same methodology as in Census Implementation 1 and split the image into 4 sections. We could obtain a frame rate of 12 fps for the same parameters as in Census Implementation. The output does not contain any noise, hence an additional filter is not required.

We see that SAD utilizes almost twice the number of LUTs than Census Implementation 1. As LUT is the most used resource, we can have only half the number of parallel SAD blocks

than Census. Thus the frame rate for SAD is half than that of Census. The reason why SAD requires more LUTs can be that we have to sum the differences between all the pixels in two windows. These arithmetic operations require more logic than the comparison with center pixel and hamming distance computation (simple bit by bit xor) done in Census Transform. Also we have to consider the width of the cost which is the sum of absolute differences between pixels in two windows. Assuming a pixel is represented by 8 bits, we consider an extreme case where the left window is all white (255) and right window is all black (0). For a window size of 11x11, the cost datatype should be at least 15 bit wide to accommodate the number 255x121. In contrast, the Census cost is sum of bits of the Hamming distance vector of length 121 (window size x window size). Hence the Census cost can be 7 bits long. This can be another reason for the increase in LUT consumption in SAD. In SGM (discussed in the next section), we store a large array of costs, thus SGM implemented using SAD as cost function will consume twice the amount of Block RAMs as compared to SGM with Census.

Figure 6.5 shows the results of census stereo matching on images from Middlebury dataset[22].



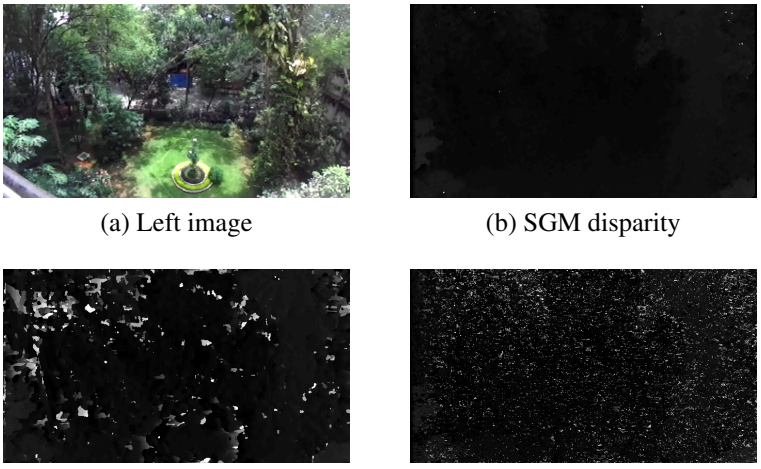
(a) Left image (b) Ground truth (c) SAD disparity image

Figure 6.6: SAD results on Middlebury images

Note: If images from the two cameras have a shift in intensity levels i.e intensities of pixels in one image have a more or less constant offset as compared to pixels in the other image, both SAD and Census will still work well. SAD algorithm will consider the sum of absolute differences of pixels in a window. Although the sum values will be higher due to offset, we will still consider the minimum of the sum values for matching. In Census Transform, all the pixels in the window are compared to the window's center pixel. As a result the effect of a constant offset is nullified.

6.3 Semi Global Matching (SGM) [20]

Census and SAD algorithms give decent results on Middlebury Dataset images. But these algorithms have to work on camera images in real time. We tested SAD, Census along with SGM on images from Zed camera. Figure 6.7 and 6.8 show the comparisons. In the garden SGM disparity image, one can make out the different layers of intensities. The disparity images from Census and SAD do contain some features but are not good. In the stairs image, we observed that both Census and SAD preserve edges in the image. The edges appear as different intensity levels even when the objects are at the same distance. This is an erroneous result. The SGM disparity image does not contain any edges. It is evident from the figures that for generating a proper disparity image from camera images, Census and SAD cannot be used. One needs sophisticated algorithms like SGM. [1]

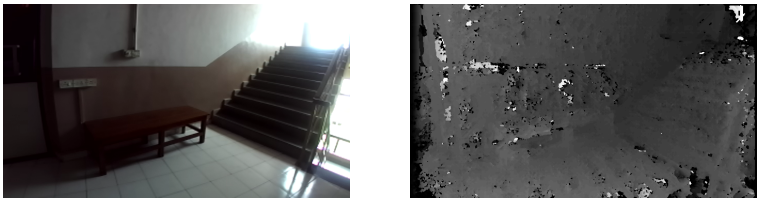


(a) Left image (b) SGM disparity

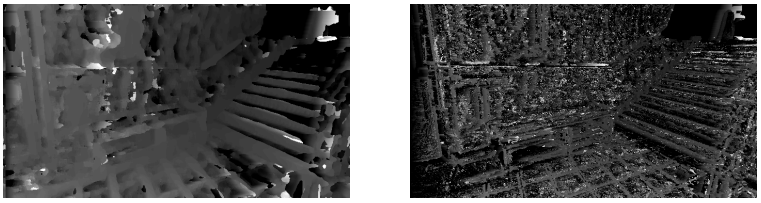
(c) SAD disparity (d) Census disparity

Figure 6.7: Stereo matching results on garden image from Zed Camera

[1] Development of SGM stereo matching algorithm for FPGA was a collaborative effort with Mr. Rashish Shingi.



(a) Left image (b) SGM disparity



(c) SAD disparity (d) Census disparity

Figure 6.8: Stereo matching results on stairs image from Zed Camera

SGM is a stereo disparity estimation method based on global cost function minimization. Various version of this method (SGM, SGBM, SGBM forest) are still among the top performing stereo algorithm on Middlebury datasets. This method minimizes the global cost function between the base image and match image and a smoothness constraint that penalizes sudden changes in neighbouring disparities. Mutual information between images, which is defined as the negative of joint entropy of the two images, is used in the paper[20] as a distance metric. Other distance metrics can also be used with similar effect as has been demonstrated with census distance metric in our implementation. Since we already had a Census Implementation, we used it for our SGM implementation. The Hamming Distance returned by Census stereo matching is used as the matching cost function for SGM. The parameters for Census are: Window size 9x9, Search range 64. It was observed that increasing the window size to 11 actually increased the noise content in the disparity image. Also decreasing the search range from 80 to 64 had no effect on the disparity image for the test scene.

Once we have fixed our cost function we can now understand the SGM algorithm. Simple census stereo matching has a cost computation step in which for a particular pixel we generate an array of costs (Hamming distances). The length of this array is equal to search range. The next step is cost minimization in which we find the minimum of this array and the index of the minimum cost is assigned as disparity. In SGM there is an additional step- cost aggregation in between cost computation and cost minimization. The computed costs are aggregated or adjusted before their minimum is found. The aggregated cost for a particular pixel p for a disparity index d is given as-

$$L'_r(p,d) = C(p.d) + min(L'_r(p-r,d), L'_r(p-r,d-1) + P_1, L'_r(p-r,d+1) + P_1, \\ min_i(L'_r(p-r,i) + P_2)) - min_k(L'_r(p-r,k)) \tag{6.1}$$

For each pixel at direction 'r', the aggregated cost is computed by adding the current cost and minimum of the previous pixel cost by taking care of penalties as shown in Equation 6.1. First term $C(p,d)$ is the pixel matching cost(hamming distance in our case) for disparity d. Before we attempt to understand the other terms, we must understand that the algorithm is recursive in the sense that to find the aggregated cost of a pixel $L'_r(p,)$, one requires the aggregated cost of its neighbours $L'_r(p-r,)$.

Since we scan through the images pixel by pixel from left to right and top to bottom, we consider only four neighbours for a pixel under processing. They are left, top left , top and top right. We will be using the aggregated costs for these adjacent neighbours to calculate the aggregated cost of the current pixel. Pixels at the top, left and right edge of the image are considered to have neighbours with maximum value of aggregated cost. As SGM cost aggregation step is a minimization function, they are effectively ignored. The 4 neighbouring paths used are shown in 6.9. The quality degradation by using 4 paths instead of 8 paths is 2-4%[21]

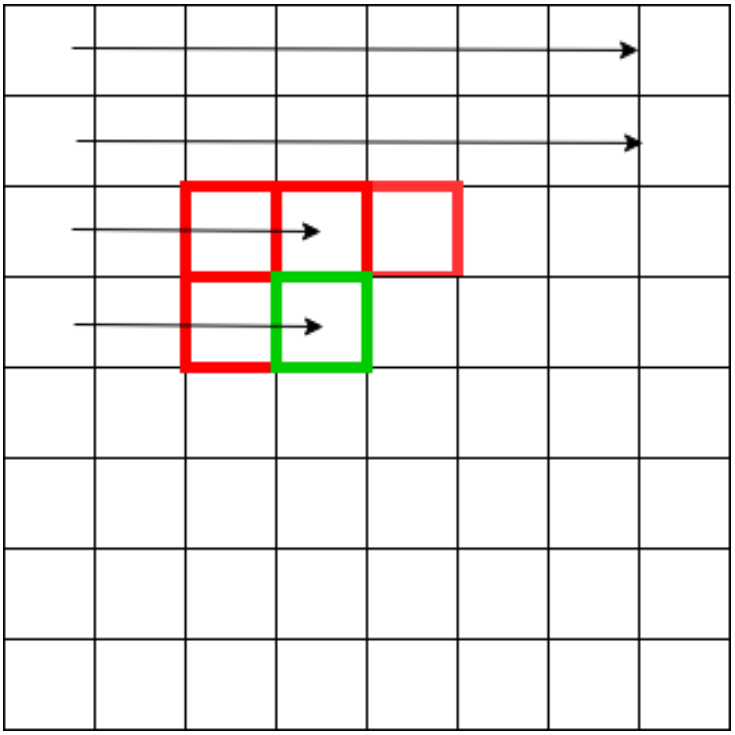


Figure 6.9: SGM using 4 neighbour paths

The term $min(L'_r(p-r,d), L'_r(p-r,d-1)+P_1, L'_r(p-r,d+1)+P_1, min_i(L'_r(p-r,i)+P_2))$ is explained using Figure 6.10. Here P1 and P2 are constant penalty values. Let's say path r=0 is the top-left path. The terms $L'_r(p-r,d), L'_r(p-r,d-1), L'_r(p-r,d+1)$ are the aggregated costs for the top left neighbour at disparity index d, d-1 and d+1 respectively. $min_i(L'_r(p-r,i)$ is the minimum of this array. These elements are shown in red in the figure. Assuming that all the required data for the neighbour pixels is available, we can compute the aggregated cost of our pixel along path 0. Similarly we find the aggregated cost of our pixel along path 1 (top), 2 (top right), 3 (left). Costs along all four paths are added to give 'sum cost'. An upper bound is applied on the sum cost. The index of the minimum of this modified sum cost gives us the disparity for this pixel.

However our task doesn't end here. Since we assumed that all the data required from neighbours was available, we must also generate that data so that it can be used by later pixels. We have to store the aggregated cost for all disparity indices along all paths. These will be 4 arrays of length equal to search range. We also compute the minimums of these arrays and store them separately. It is possible to not store the minimum of arrays and compute the minimum every time it is required. But this will cause unnecessary computations. At this point we have generated the data which we assumed was available from our neighbours. We will now see how to

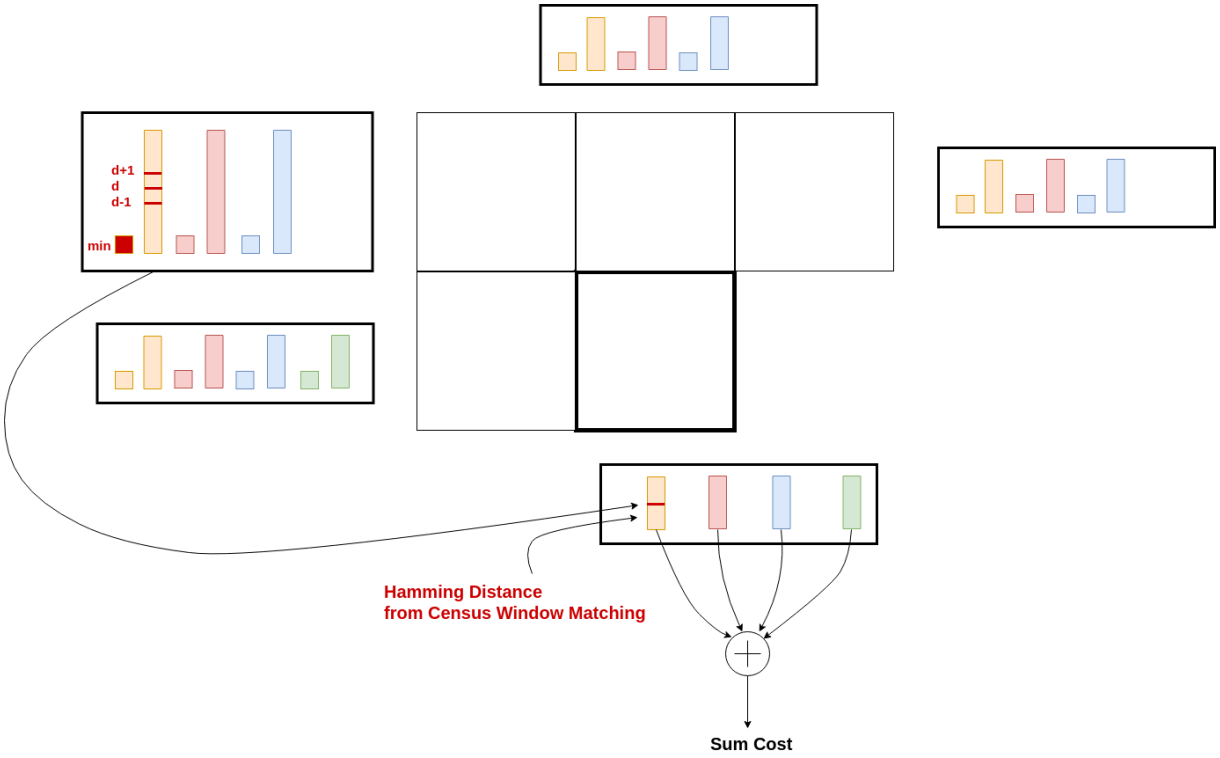


Figure 6.10: SGM Cost Computation

store this data.

We store the aggregated cost for all paths for every disparity index of one row above the pixel and the left adjacent row of the pixel under process. Minimum cost (for all paths) across search range is also stored for the above row and left adjacent pixel. This is shown in Figure 6.11 a using an example. Suppose currently the pixel under process is at row 6 col 20. For this pixel we will need aggregated costs of pixels at: row 5 col 19 (top left), row 5 col 20 (top), row 5 col 21 (top right) and row 6 col 19 (left) which are stored in the $cost\_row$ and $cost\_left$ data structure. Once we are done processing this pixel and generating the data required for later pixels, the data stored at index 19 of $cost\_row$ (row 5 col 19) is obsolete as it will not be required for any future pixel. The data from $cost\_left$ is moved to index 19 of $cost\_row$. Then the data from current pixel is moved to $cost\_left$. Now we are ready to move to the next pixel. This is shown in Figure 6.11 b. We can also see that the new pixel under process row 6 col 21 has all the data required from the neighbour pixels. It can be seen that following this process, we can scan through all the pixels in the image and generate the disparity image.

We have to store aggregated costs along all paths in the $cost\_row$ structure. This is because though row 6 col 20 pixel used data from path 1 (top) of row 5 col 20, but for the next pixel row

6 col 21, the pixel at row 5 col 20 is the top left neighbour and hence it requires data from path 0 (top left) of row 5 col 20. One important optimization is that we need to store aggregated costs along 3 paths in the *cost_row* structure. This is because the fourth path- left is not required. Hence while updating *cost_row* from $cost\_left$, the data for left path is not updated.

We mentioned earlier that pixels at the top, left and right edge of the image are considered to have neighbours with maximum value of aggregated cost. This is done by initializing *cost_row* and $cost\_left$ structures with maximum cost values.

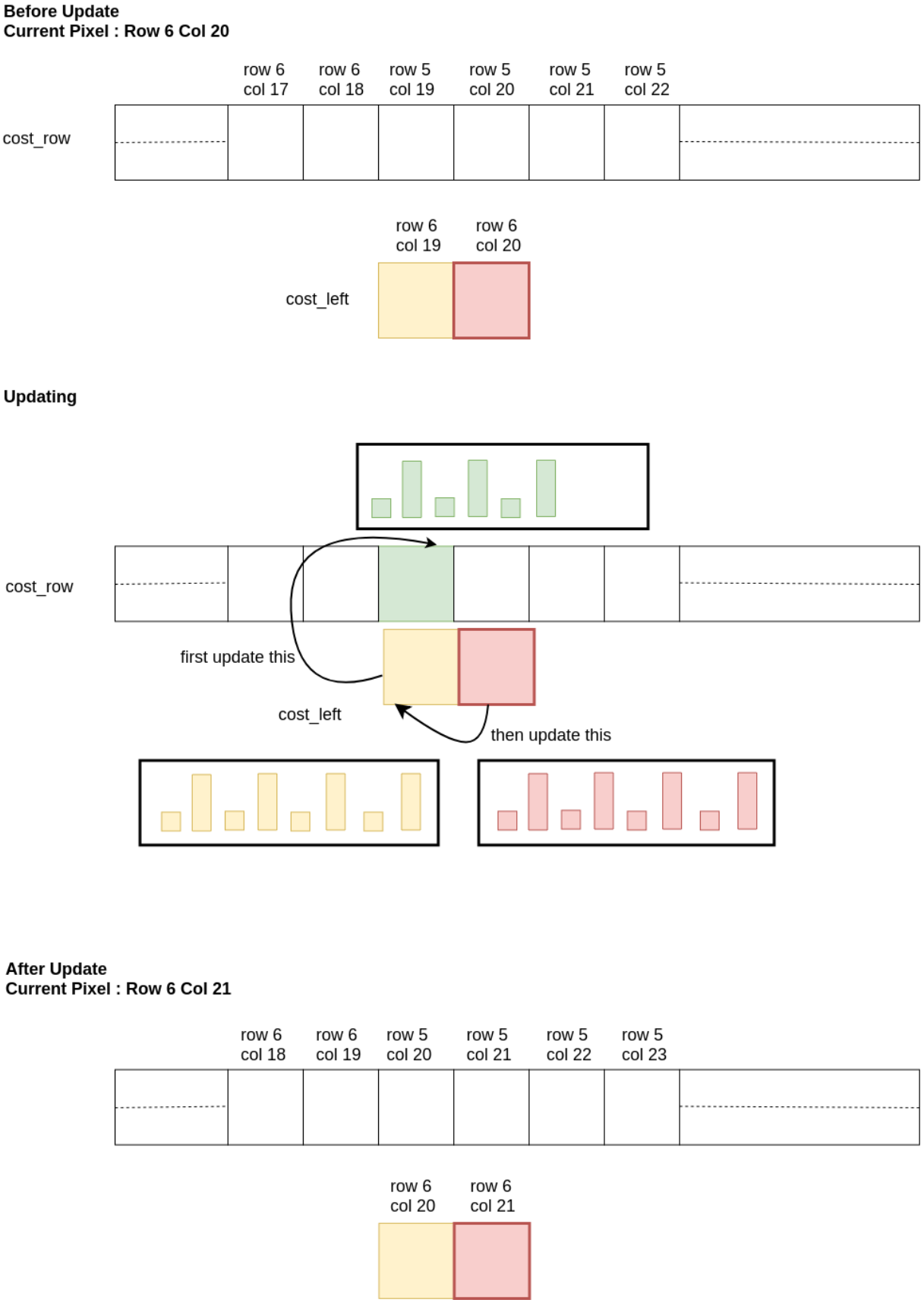
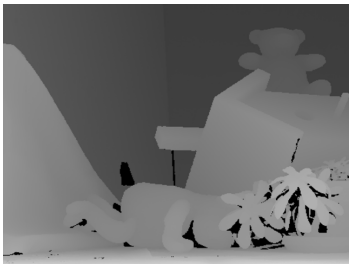


Figure 6.11: SGM Array Updation

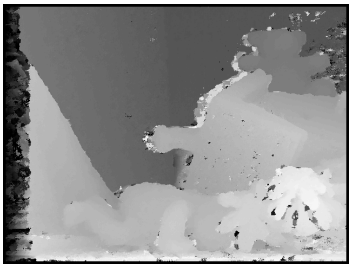
Following the methodology in Census Implementation 2, we tried to modify the second peripheral which computes disparity from Census Vectors. But the resources in Zedboard were not enough, so the search_range loop in SGM could not be unrolled. So we had to resort to using the methodology in Census Implementation 1. It is worth notable that SGM uses 34% BRAMs as compared to 8% for Census. This is because we need to store a lot of data of the neighbors of the pixel under process. In the SGM implementation BRAM (34%) is the most contented resource. We tried to use 3 SGM blocks, but the design did not fit in Zedboard. So we had to go for 2 blocks which gave us a frame rate of 4 fps. Although the storage overhead for SGM is quite large, the computation overhead is only 60% more as compared to Census. That means given the required BRAMS for SGM and keeping all other resources same, SGM will take 1.6 times the time required to process data than Census. This means, if we optimize the underlying Census we will also be automatically optimizing SGM.

The SGM implementation was compared to a software implementation of SGM[5]. Figure 6.12 shows the software and hardware implementation results on Middlebury images. We can observe that SGM with 8 paths gives the best results. SGM with 4 paths in software gives slightly better results than the hardware implementation. The difference in results is due to the fact that the way the algorithm is implemented in software and hardware is different. Figure 6.12 f shows the SGM disparity image with *cost_row* and $cost\_left$ initialized to zero. Since the cost aggregation function is minimization function, the zeros from the arrays propagate to further pixels. The trickle down effect causes the degradation of the disparity image.

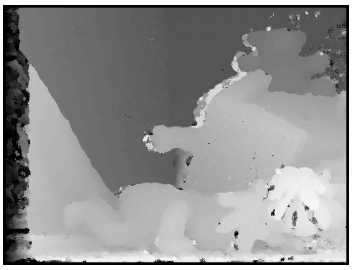
(a) Left image



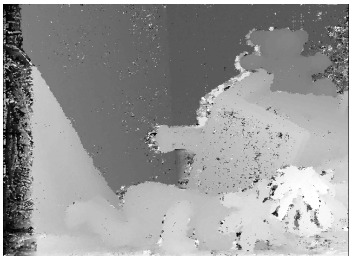
(b) Ground truth



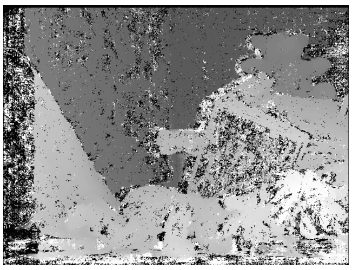
(c) SGM 4 paths software



(d) SGM 8 paths software



(e) SGM 4 paths hardware



(f) SGM with arrays initialized to zeros

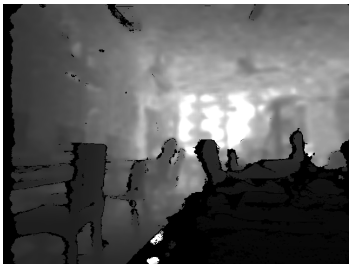
Figure 6.12: SGM results on Middlebury images

The SGM implementation was tested on images captured from Intel Realsense D435i camera. The depth image was generated using our SGM hardware implementation. The realsense camera itself computes the depth image using the onchip processor. We have also captured that depth image. It is compared to the SGM depth image. Figures 6.13 shows a lab scene. The images were captured with the IR blaster on. The IR pattern can be observed on the wooden table in subfigure (a) and (b). Subfigure (c) shows the intel depth image, (d) shows the SGM hardware depth image and (e) shows the SGM (with 4 paths) software depth image. Notice that for the intel depth image, the convention followed is that far away objects have higher intensity. It can be observed that SGM performs reasonably well but has some noise. Figure 6.14 and 6.15 show the effect of IR blaster. Images in Figure 6.14 were captured with the IR blaster on whereas images in Figure 6.15 were captured with the IR blaster covered. The IR pattern is clearly visible in Figure 6.14. We observed that the pattern stayed visible in the images to human eyes for objects upto a distance of 3 meters. The depth image in 6.14 contains some salt noise, but it can be easily removed. One can observe the fan blades clearly. The depth image is severely degraded when the IR blaster is covered. Although we can clearly make out the fan, it will be difficult to filter out the noise in this image. Thus we can conclude that the IR pattern helps in stereo matching by adding texture to non-textured surfaces. We later found in the intel realsense forum[14] out that this is indeed true.

Figure 6.16 and 6.17 shows the SGM disparity image for ELP camera and Zed camera respectively. It can be observed that SGM gives decent results for Zed camera even though no structured light is involved. The results for ELP camera are bad maybe because the images were not rectified correctly. Zed Camera calibration parameters are available to download from the developer's website but ELP camera had to be calibrated by us.

(a) Left image

(b) Right image

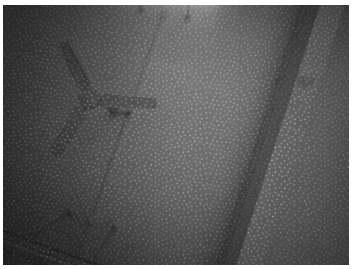


(c) Intel depth image

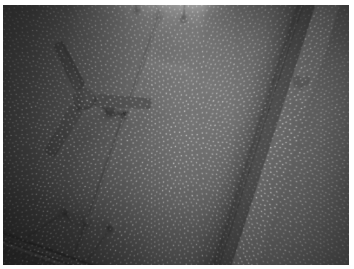
(d) SGM hardware depth image

(e) SGM software depth image

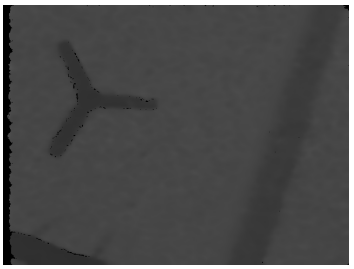
Figure 6.13: SGM results on Realsense image: lab



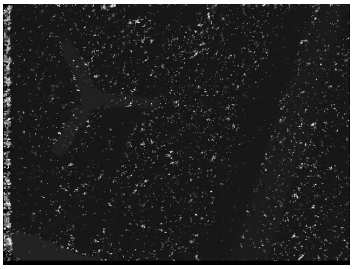
(a) Left image



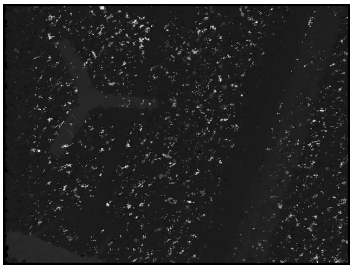
(b) Right image



(c) Intel depth image



(d) SGM hardware depth image

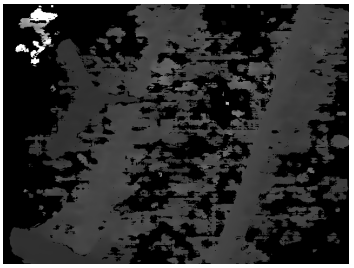


(e) SGM software depth image

Figure 6.14: SGM results on Realsense image: fan with IR blaster on

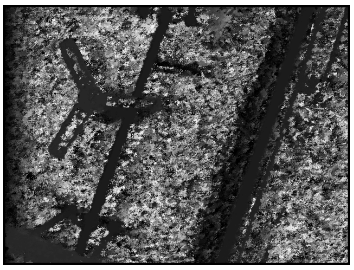
(a) Left image

(b) Right image



(c) Intel depth image

(d) SGM hardware depth image

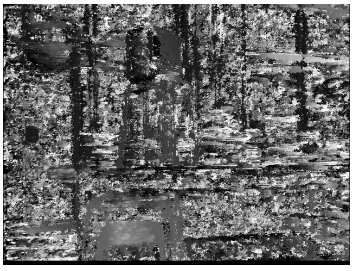


(e) SGM software depth image

Figure 6.15: SGM results on Realsense image: fan with IR blaster covered

(a) Left image

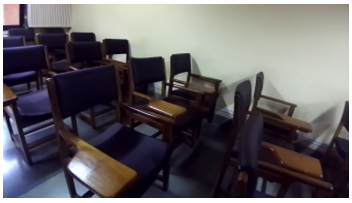
(b) Right image



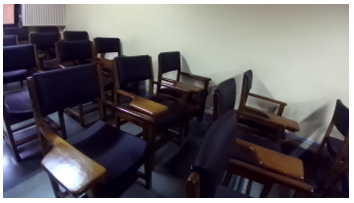
(c) SGM hardware depth image

(d) SGM software depth image

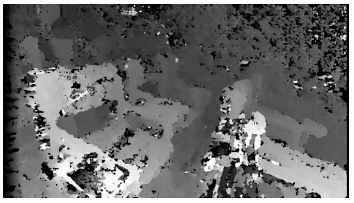
Figure 6.16: SGM results on ELP image: stool



(a) Left image classroom



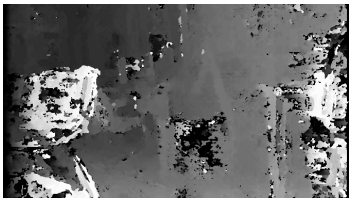
(b) Right image classroom



(c) SGM depth image classroom

(d) Left image lab

(e) Right image lab



(f) SGM depth image lab

Figure 6.17: SGM results on ZED camera images

Chapter 7

Observations and Conclusion

7.1 Stereo matching process

Stereo matching cannot be used in single intensity planar regions as there will be multiple windows with a high correlation with the reference window. However presence of periodic texture in the image can spoil the depth image due to incorrect matching. This can be observed in Fig 6.6 c, where the texture of the house roof creates distortions in the depth image. The disparity image quality can be improved by using a structured light projector[14]

Image resolution, window size and disparity search range are important parameters for a stereo matching algorithm. Window size can be estimated from the resolution alone but search range requires a knowledge of the expected disparity. For example knowing that the objects will be placed very close to the camera baseline or cameras are placed far apart, the search range will be increased to give correct results.

7.2 Stereo matching algorithms

SAD and Census cannot be used for general images captured from stereo cameras. SGM or more sophisticated algorithms are required for obtaining decent disparity images.

7.3 Hardware

SGM frame rate is limited by number of Block RAMs available in Zedboard. If we use a FPGA chip with more amount of Block RAMs, its frame rate can be increased. Frame rate

is proportional to clock frequency of the FPGA. It can be increased by using a chip which supports higher frequencies. USB 3.0 is required for interfacing Intel RealSense D435i, but is not available in Zedboard. Also USB 3.0 is required for capturing uncompressed images from left and right camera simultaneously. USB 2.0 does not have enough bandwidth to do the same.

Chapter 8

Future work

- Generation of 3D depth map from disparity images.
- USB3 Vision core[23] is an IP available for capturing images from USB cameras. It can be integrated into the project once a FPGA board compatible with USB 3 is available.
- SGM can be implemented with 1,2 or 3 paths, so as to find a good tradeoff between disparity image quality and computation time.
- Connect the first and second peripheral in Census Implementation 2 using FIFOs.
- Handcraft the second peripheral of Census Implementation 2 which computes disparity from census vectors. Once that is done, we can modify it for SGM.
- Variants of SGM like Parametric SGM, Weighted SGM, Iterative SGM can be tried to improve its quality.
- Post processing can be done on SGM output to improve its quality.

Chapter 9

Appendix

9.1 Petalinux

9.1.1 Installation

- We have `Vivado 2017.1`, so we chose to install `Petalinux-v2017.1`
- Download `petalinux-v2017.1-final-installer.run` from Xilinx website. (Requires xilinx account which can be easily made)
- Make a new directory in which petalinux will be installed.
  `mkdir /opt/Xilinx/petalinux-v2017.1/`
- Go to that directory. `cd /opt/Xilinx/petalinux-v2017.1/`
- The downloaded file is an executable. We may not have the execute permission for it so we will give it. `chmod +x ~/Downloads/petalinux-v2017.1-final-installer.run`
- Now we will execute the file in this directory.
  `~/Downloads/petalinux-v2017.1-final-installer.run`
- The installer may notice few missing dependencies. They can be installed using apt. `libssl-dev` is required to forego the `libssl` dependency.

9.1.2 Building a Petalinux project

- Create a Vivado Project with Zynq PS and the required peripherals. In the Vivado Project, `File -> Export Hardware`. This will produce a `.hdf` file in `<project name>.sdk directory`.

- To use petalinux, source following file. `/opt/Xilinx/petalinux-v2017.1/settings.sh`
- Create a new petalinux project.
  `petalinux-create --type project --template zynq --name <project name>`
- This will create a directory with the given project name. Go to that directory. `cd <project name>`
- Provide the hardware description from the Vivado project sdk folder.
  `petalinux-config --get-hw-description=<path/to/vivado_project.sdk/>`
  This opens a GUI, in which we have to disable following option.
  `petalinux-config -> Yocto settings -> Enable Network sstate feeds`
  Otherwise it takes forever to compile the project.
- Configure the kernel to enable UVC inputs.
  `petalinux-config -c kernel`
  A GUI opens up in which following steps have to be done-
  Device Drivers -> Multimedia support
  `[*] Media USB Adapters`
  `Device Drivers -> Multimedia support -> Media USB Adapter`
  `<*> USB Video Class (UVC)`
  `[*] UVC input events device support (NEW)`
- The file `project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi` has to be appended with some lines to enable ethernet, USB device support (although not needed in this project) and reserving some memory in DDR RAM. We have a working `system-user.dtsi` file available which can replace the existing file in the project. The contents of the file are explained below.
  Enabling ethernet:

```
/include/ "system-conf.dtsi"
/ {
};

&gem0 {
```

```
        phy-handle = <&phy0>;
        phy0: phy@0 {
            compatible = "marvell,88e1510";
            device_type = "ethernet-phy";
            reg = <0x0>;
            marvell,reg-init = <0x3 0x10 0xff00 0x1e 0x3 0x11 0xfff0 0xa>;
        };
};
```

Enabling USB device support:

```
/{
    usb_phy0:phy0 {
       compatible="ulpi-phy";
       #phy-cells = <0>;
       reg = <0xe0002000 0x1000>;
       view-port=<0x170>;
       drv-vbus;
    };
};

&usb0 {
        status = "okay";
        dr_mode = "host";
        usb-phy = <&usb_phy0>;
};
```

Reserving space in DDR memory. This space is reserved for storing images (left, right, disparity) and the generated rectification maps. Zedboard has 512MB RAM (Address Range 0x0 to 0x1FFFFFFF). Out of this some memory at the beginning will not be available to use for us. Also it is observed that 15MB gets allocated near the end of memory for 'cma'. Hence it will be better if we reserve some memory excluding such extreme addresses. In the following snippet we have reserved 32MB (0x1000000) memory starting

from location 0x1a000000.

```
/{
    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        reserved: buffer@0x1a000000 {
            compatible = "xlnx,zynq-7000";
            reg = <0x1a000000 0x1000000>;
        };
    };
};
```

- The next step is to build the petalinux project.
  `petalinux-build`

- Here we pass the bitsream from the Vivado project and package the petalinux project. The following line is a single command.
  `petalinux-package --boot --fsbl images/linux/zynq_fsbl.elf --fpga /path/to/Vivado_project/bitstream.bit --u-boot`

- Copy files to bootable partition of sd card
  `cp BOOT.BIN ./images/linux/image.ub sd-card/bootable/partition/`

Note: Guide to prepare sd card for sd card boot is available at
https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842385/How+to+format+SD+card+for+SD+boot

9.1.3 Booting Zedboard with Petalinux

- Insert sdcard in the zedboard.

- Keep jumper settings as: MDIO6,5,4,3,2 => 0,1,1,0,0
  1 indicates connection between 3V3-SIG and 0 indicates connection between SIG-GND

- Power-on the board. The blue led should light up indicating that the board is programmed.
- Open minicom serial connection.
  `minicom -s -D /dev/ttyACM0 -b 115200`
- Press any key, a `Zynq>` prompt should appear. Type `boot` at the `Zynq>` prompt.
- The board boots up. The log can be observed to look for reserved memory and ethernet availability.
- Once the boot process is complete, we get a username and password prompt. Both are `root`. Once login is done, we should get a `root@plnx_arm` prompt.
- The bootable partition appears as `/dev/mmcblk0p1`.
  To mount the partition: `mount /dev/mmcblk0p1 /mnt/`
- We can go into `/mnt` directory now and list its contents. We will observe `BOOT.BIN` and `image.ub` files.
- As we have ethernet enabled, we can simply scp (copy over network) the new `BOOT.BIN` and `image.ub` into /mnt/ and type `reboot` to boot the Zedboard with new petalinux. For that we require ip-address. The ip address assigned by default will be random so we will fix it so that we can scp easily.
  To fix the ip address to a fixed value. `ifconfig eth0 10.107.88.118`
- Mounting and setting the ip address has to be done every time the board is powered on.

9.1.4 Installing Opencv for Petalinux

All lines are single line commands.

- `mkdir opencv_setup`
- `cd opencv_setup`
- `Download zip from https://github.com/opencv/opencv.git`
- `unzip opencv-master.zip`
  (We used opencv 4.0.1)

- `mkdir build`
- `cd build`
- `export PATH=$PATH:`
  `/opt/Xilinx/petalinux-v2017.1/tools/linux-i386/gcc-arm-linux-gnueabi/bin/`
- `cmake -DCMAKE_TOOLCHAIN_FILE=../opencv-master/platforms/`
  `linux/arm-gnueabi.toolchain.cmake ../opencv-master/`
- `make`
- `make install`
- `ls install`

Note: Steps till this point may also be available at local machine:
`http://10.107.88.22/Misc/petalinux.html`

9.1.5 Cross compiling application programs

- arm-linux-gnueabihf-g++ cross compiler is used to cross compile programs for ARM processor. It is from Petalinux, hence `path/to/installation/petalinux-v2017.1/settings.sh` has to be sourced for using it.
- The compiled binaries of standard libraries, openmp and opencv libraries can be kept in a directory in /mnt/ eg. If we keep these files in /mnt/lib , while compiling we have to provide the flag -rpath=/mnt/lib in the linking step or we have to add the path to this directory to `LD_LIBRARY_PATH` in Zedboard.
- We have made a Makefile to ease this process. As we have to send the generated executable to Zedboard, we have added the scp step in the Makefile itself.

```
APP_NAME=<program_name>
.PHONY: $(APP_NAME).elf
$(APP_NAME).elf: $(APP_NAME).cpp
#compile
arm-linux-gnueabihf-g++ -Wall -O3 -g3 -c -fmessage-length=0 -fopenmp
```

```
-o $(APP_NAME).o -I /..../opencv/build/install/include/opencv4 $(APP_NAME).cpp
#link
arm-linux-gnueabihf-g++ -O3 -Wl,-rpath=/mnt/lib -fopenmp -o $(APP_NAME).elf
$(APP_NAME).o -L/..../opencv/build/install/lib -lopencv_calib3d -lopencv_core
-lopencv_dnn -lopencv_features2d -lopencv_flann -lopencv_gapi-lopencv_highgui
-lopencv_imgcodecs -lopencv_imgproc -lopencv_ml -lopencv_objdetect
-lopencv_photo -lopencv_stitching -lopencv_videoio -lopencv_video
scp -r -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null
$(APP_NAME).elf root@10.107.88.118:/mnt/
clean:
rm $(APP_NAME).elf $(APP_NAME).o
```

- Once the elf file is in Zedboard, we have to just execute it.

9.1.6 Structure of main application program

The application program tasks can be classified as

- continuously capturing data from camera
- memory access

The memory access step involves setting up the peripherals (moving appropriate data into different peripheral registers), storing rectification maps into DDR and starting the peripherals (moving data into peripheral control registers). The memory access step has to be done only once. Peripherals can be started on autorestart i.e they start after they have finished processing. Hence we first perform the memory access steps and then proceed into an infinite while loop for capturing images continuously.

9.2 Vivado Project

Figure 9.1 shows the block diagram for the final implemented project. It has Zynq PS block along with its reset logic. We have included the following FPGA peripherals: 2 SGM blocks, 2 remap blocks for left and right image respectively and VGA block for displaying the image. The AXI interconnect and smartconnect peripherals were added by connection automation. It

is important to note that we had to use 2 High performance HP PS-PL slaves for connecting our peripherals as a single HP slave gave erroneous results.

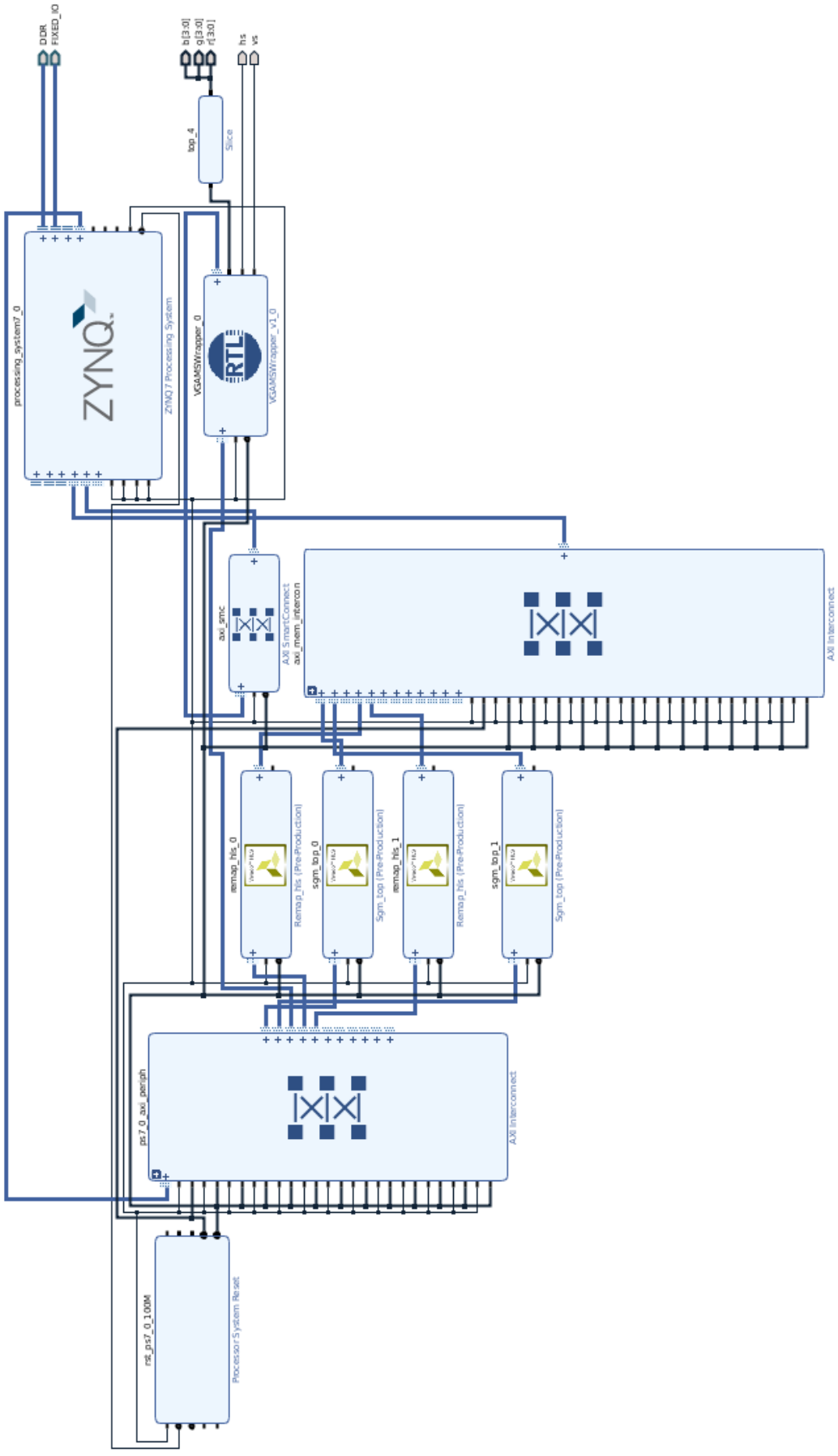


Figure 9.1: Vivado project with 2 remap blocks (left, right), 2 SGM blocks and VGA peripheral

9.3 Teraranger Duo [25]

Teraranger Duo is a Time of Flight (ToF) and Ultrasound based distance measurement sensor. ToF has a high update rate of 1 KHz, but may have problems with certain materials like glass. In these cases, Ultrasound can detect those surfaces well though at a slower update rate of 10Hz. It does not provide dense depth data from the surrounding. It provides the distance from a obstacle point in a 3.4 degree field of view in front of it. Hence it cannot be used for constructing depth maps. However it can be used as an auxiliary sensor which can be mounted at the bottom of an aerial platform to keep it at a safe flying height. It is light weight (16g), has UART interface for reading the data and also has ROS module under development. A limitation of the sensor is that it requires 10V power supply.

We used a PL2303 USB UART converter for interfacing the Teraranger Duo to a PC. The setup is shown in Figure 9.2. A python script was used for reading data from the sensor.

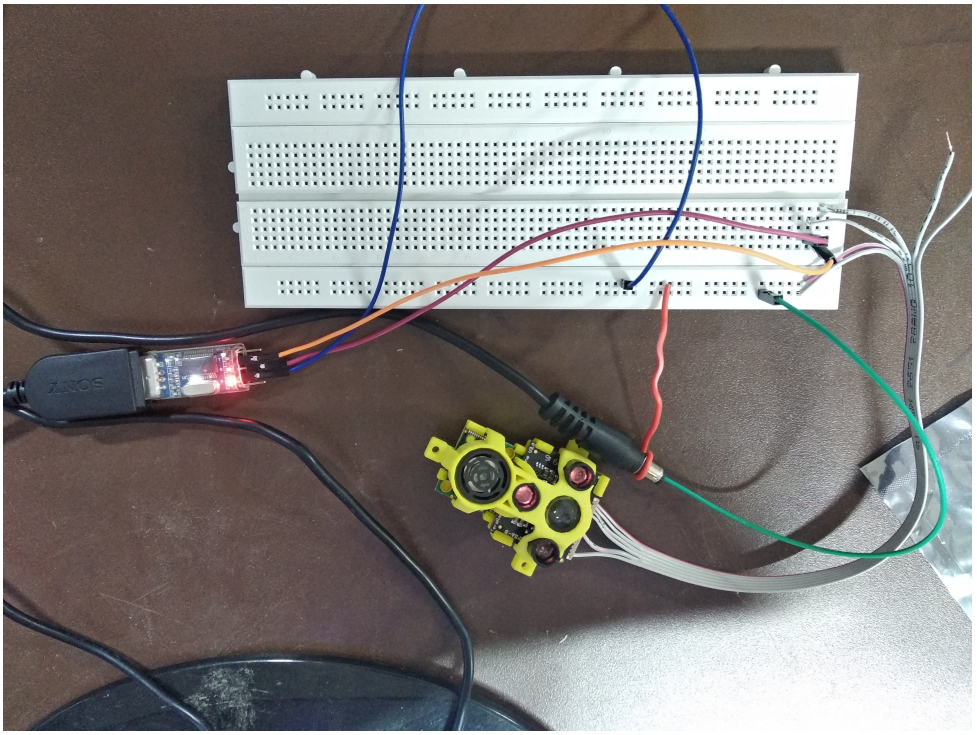


Figure 9.2: Teraranger Duo setup

Bibliography

[1] Grandhi Kiran Kumar, FPGA based implementation of depth perception from stereo vision, M.Tech Dissertation Report.

[2] Chandan M L, Environment development for stereo vision with software-hardware interface using Zedboard, M.Tech Dissertation Report.

[3] Nikhar Gangrade, Stereo Image Rectification Module implemented on FPGA, M.Tech Dissertation Report.

[4] Rashish Shingi, Depth Map Estimation and Real-Time Implementation on FPGA, M.Tech Dissertation Report.

[5] Imran Syed, Acceleration of Stereo Vision Algorithms using Jetson TK1, M.Tech Dissertation Report.

[6] Zedboard datasheet:
http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[7] Zynq 7000 datasheet:
https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[8] Vivado HLS user guide:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[9] Vivado Synthesis user guide:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado-synthesis.pdf

[10] XSCT reference guide:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug1208-xsct-reference-guide.pdf

[11] OV7670 datasheet:
https://www.voti.nl/docs/OV7670.pdf

[12] ELP Camera:
https://www.amazon.in/ELP-Megapixel-Camera-Biometric-Retinadual/dp/B00VG32EC2

[13] Intel Realsense D435i Depth Camera:
https://www.intelrealsense.com/depth-camera-d435i/

[14] https://forums.intel.com/s/question/0D50P0000490VMfSAM/d435-questions?language=en_US
"The primary means of depth detection is through stereo vision technology and is assisted by an IR projector on the module that provides texture through structured light."

[15] Zed Camera:
https://www.stereolabs.com/

[16] W.Daolei, K.B.Lim, Obtaining depth maps from segment based stereo matching using graph cuts, in J.Vis.Commun. Image R.22 (2011)325-331.

[17] HLS Pragmas:
https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uyd1504034366571.html

[18] R. Zabih and J. Woodfill [*Non-parametric local transforms for computing visual correspondance*] In Proc. ECCV, pages 151–158, 1994

[19] T. Kanade [*Development of a video-rate stereo machine*] In IUW , pp. 549-557, 1994

[20] H. Hirschmuller [*Stereo Processing by Semiglobal Matching and Mutual Information*]

[21] M. Roszkowski and G. Pastuszak [*FPGA design of the computation unit for the semi-global stereo matching algorithm*] doi: 10.1109/DDECS.2014.6868796

[22] D. Scharstein and R. Szeliski [*High-accuracy stereo depth maps using structured light*] In IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003), volume 1, pages 195-202, Madison, WI, June 2003

[23] USB 3 Vision Core:
https://www.xilinx.com/products/intellectual-property/1-4vci8x.html#productspecs

[24] Bradski, G. and Kaehler, A. (2008). Learning OpenCV. OReilly Media. ISBN 978-0-596-51613-0.

[25] Tera Ranger Duo: http://www.teraranger.com/product/teraranger-duo/