Implementation and Analysis of Stereo Vision Algorithms for different FPGAs

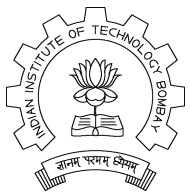
Dual Degree-Project Stage 1

by

T Yashwant Kumar
Roll No: 15D070056

under the guidance of

Prof. Sachin Patkar



Department of Electrical Engineering
Indian Institute of Technology, Bombay
Mumbai - 400076.

2019

Abstract

Perception of the environment is the key component for any task in the robotics or any industrial automation. For this, one have to precisely map the objects in the 3D space for the bot to be able to interact with them. Extracting this accurate depth information of the surrounding is possible through some sensors like LIDAR(Light Detection and Ranging), Time of Flight sensor or a Stereo Camera. One of the most popular approach is via computation of disparity-map of images obtained from Stereo Camera. This work discusses a low power, high performance implementation of the complete system on an FPGA which computes the disparity image using images captured from a Stereo Camera and generates a 3D Map. Semi Global Matching (SGM) method is a popular choice for good accuracy with reasonable computation time. To use such compute-intensive algorithms for real-time applications such as for autonomous aerial vehicles, blind Aid, etc. acceleration using GPU, FPGA is necessary. We discuss here the design and implementation of a stereo-vision system, which is based on FPGA-implementation of More Global Matching(MGM)[7]. MGM is a variant of SGM. We use 4 paths but store a single cumulative cost value for a corresponding pixel. Our stereo-vision prototype uses Zedboard / PYNQ / Ultra96 containing an ARM-based Zynq-SoC, ZED Stereo Camera / ELP Stereo Camera. The power consumption attributed to the custom FPGA-based acceleration of disparity map computation required for depth-map is 0.72 watt on Zedboard. The update rate of the disparity map is realistic 10 fps on Zedboard / PYNQ and 16 fps on Ultra96 .

Contents

List of Figures

Chapter 1

Introduction

For mapping of environment although 2D and 3D LIDARs (Light Detection and Ranging Sensors) provided accuracy, they did not succeed with the economics of power and bill of materials for portable goods. Stereo cameras cost less, but need a lot of computational processing, and this aspect is getting good attention of research community , spurring the development of FPGA and GPU based acceleration of stereo-vision related computation. The low power consumption of fpga-based solutions are attractive and crucial for high performance embedded computing too.

This idea of stereo matching is similar to the way we perceive the surroundings with our eyes. Our brain receives 2 views from 2 eyes with a lot of similarities in them but with few differences, of which the linear shift in a particular object corresponds to the information of the how far the object is from us. The objects closer will have a higher shift and the objects far from us will have less.

This work describes our design and implementation of a real-time stereo depth estimation system with Zedboard / PYNQ / Ultra96 (housing ARM-SoC based FPGA) at its center. This system uses Zed stereo camera[16] or ELP stereo-camera for capturing images. Real-time Raster-Respecting Semi-Global Matching[6] (R3SGM) with MGM[7] at its core along with Census Transform are used for disparity estimation. The system takes in real-time data from the cameras and generates a depth image from it. Rectification of the images, as well as stereo matching, is implemented in the FPGA[22] whereas capturing data from USB cameras and controlling the FPGA peripherals is done via application programs which run on the hard ARM processor on Zedboard. Development of the FPGA IP's is done using High-Level Synthesis (HLS) tools.

Chapter 2

Literature Review

There has been a lot of research on the topic of disparity map generation dating back to 1980s. [8] reviews most of the works including both software and hardware implementations.

A binocular Stereo Camera estimates disparity or the difference in the position of the pixel of a corresponding location in the camera view by finding similarities in the left and right image. There have been various costs governing the extent of the similarity. Some of them are Sum of Absolute Differences(SAD), Sum of Squared Differences(SSD), Normalized Cross-Correlation and the recent Rank Transform and Census Transforms. They are window-based local approaches where the cost value of a particular window in the left image is compared to the right image window by spanning it along a horizontal axis for multiple disparity ranges. The window coordinate for which the metric cost is the least is selected which gives us the disparity for that corresponding center pixel. From the disparity, the depth value is computed by equation 2.1 where the baseline is the distance between the optical centers of two cameras and uses focal length of the camera.

$$Depth = Baseline * (FocalLength)/disparity \tag{2.1}$$

Local window-based approaches suffer when the matching is not reliable which mostly happens when there are very few features in the surrounding. This results in the rapid variations of the disparities owing to a less accurate map. This problem is solved by global approaches which use a smoothing cost to penalize wide variations in the disparity around a pixel and try to propagate the cost across various pixels. The following are some of the global approaches.

2.1 SGM

SGM is a stereo disparity estimation method based on global cost function minimization. Various versions of this method (SGM, SGBM, SGBM forest, MGM) 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 neighboring disparities. Mutual information between images, which is defined as the negative of joint entropy of the two images, is used in the paper[3] as a distance metric. Other distance metrics can also be used with a similar effect as has been demonstrated with census distance metric in our implementation. Since we already had a Census Implementation[21], 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 7x7, disparity search range 92. The image resolution is 640x480.

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 the disparity search range. The next step is cost minimization in which the minimum of this array (minimum cost) is computed and the index of the minimum cost is assigned as disparity. In SGM, an additional step of cost aggregation is performed between cost computation and cost minimization. The aggregated cost for a particular pixel p for a disparity index d is given by equation 2.2.

$$\begin{aligned} L_r(p,d) = C(p.d) + min(&L_r(p-r,d),\\ &L_r(p-r,d-1)+P_1,\\ &L_r(p-r,d+1)+P_1,\\ &min_i(L_r(p-r,i)+P_2))\\ &-min_k(L_r(p-r,k)) \end{aligned} \tag{2.2}$$

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 2.2. First-term $C(p,d)$ is the pixel matching cost for disparity d. In our case, it is the Hamming distance returned by Census window matching. It is apparent 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 neighbors $L'_r(p-r,)$. P_1 and P_2 are empirically determined constants. For detailed discussion refer to [3].

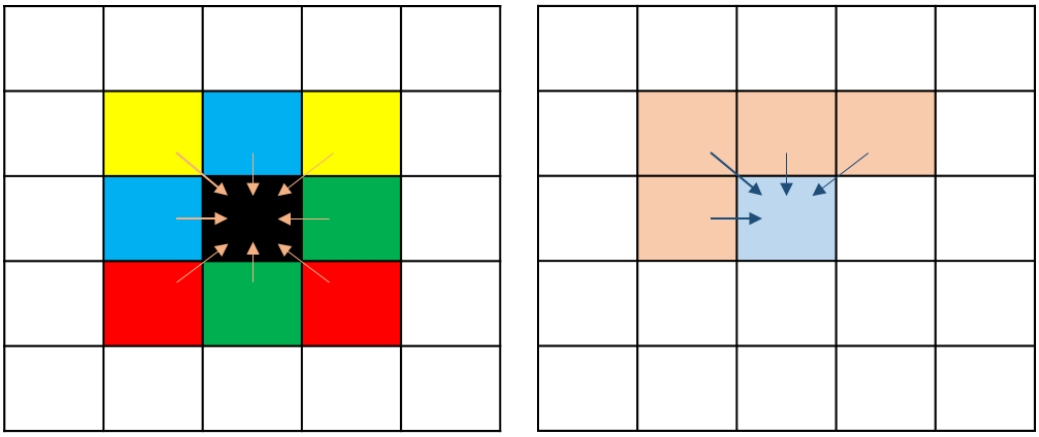
2.2 MGM

As SGM tries to minimize the cost along a line it suffers from streaking effect. When there is texture less surface or plane surface the matching function of census vector may return different values in two adjacent rows but due to SGM, the wrong disparity may get propagated along one of the paths and can result in streaking lines.

MGM[7] solves this problem by taking the average of the path cost along 2 or more paths incorporating information from multiple paths into a single cost. It uses this result for the next pixel in the recursion of Equation 2.2. The resultant aggregated cost at a pixel is then given by the Equation 2.3

$$\begin{aligned} L_r(p,d) = C(p.d) + 1/n \sum_{x\varepsilon\{r_n\}} (&min(L_r(p-x,d),\\ &L_r(p-x,d-1)+P_1,\\ &L_r(p-x,d+1)+P_1,\\ &min_i(L_r(p-x,i)+P_2))\\ &-min_k(L_r(p-x,k))) \end{aligned} \tag{2.3}$$

where n has the value depending on the number of paths that we want to integrate into the information of single cost. For example, in Figure 2.1a two paths are grouped into 1 so n has value 2 and there are a total of 4 groups. Thus we need to store 4 cost vectors in this case and while updating 1 cost value in the center pixel have to read cost vector of the same group from 2 pixels. Lets say $r = 1$ for blue boxes group in Figure 2.1a, while updating the L_r for this group of the centre pixel in Equation 2.3 we have x as left and top pixels. Similarly it can be understood for our implementation2.1b where $n = 4$ and only 1 group of 4 paths is used resulting in only 1 vector to be stored. From here on SGM refers to MGM variant of it.



(a) MGM in General (b) Our Implementation

Figure 2.1: Grouping of Paths in MGM

Chapter 3

Hardware architecture and Implementation

3.1 System Design

Figure 3.1 shows an overview of the implemented system. Using the Processing System(ARM) raw images are captured through USB. Cameras used in this system transmit video feed through Universal Video Class(UVC). For the ELP camera the 2 images left and right stream in 2 different video ports i.e. video0 and video1 on the board which is connected to a single USB2.0 bus. Because of capturing 2 images at 2 different times there is synchronisation problem and we compare the left frame with right frame captured few milli seconds later. This causes a slight decrement in the accuracy of the disparity image. Zed camera transmits the left and right images in a single stitched frame which removes the issue of synchronisation between the frames. For an image with resolution of 640*480 it transmits a frame of 1280*480 through a single video feed video0 where left half of the frame is left image and vice versa.

We store these images into some location of the DRAM. Rectification of the stored images is then done to remove the distortion effect due to the wide angle lens. [22] has implemented a hardware IP for rectification. This has been used to rectify the ELP camera images. Rectification uses ReMap Matrices to compute the transform. These matrices are computed offline for once by calibration functions of opencv library using a checker board. The resultant 2 matrices for 2 images respectively are read from a .yml file and stored into the reserved space in dram. This Rmap data is used by the ReMap IP and it stores the computed Rectified images back in the DRAM. For Zed camera the rectification is done in the Application program itself on ARM processor. For this Remap function of OpenCV is used.

These rectified images are read by the SGM block which computes the disparity image and stores the result. The VGA peripheral continuously displays the data read from the location where Depth Image is stored. Data transfer between FPGA and DRAM is done using AXI protocol where the IPs are configured to be as master axi. The PS takes care of the memory read write requests from the IP and also initates the peripherals to start the computation.

The resolution of images is fixed to 640x480 and cameras are configured accordingly.

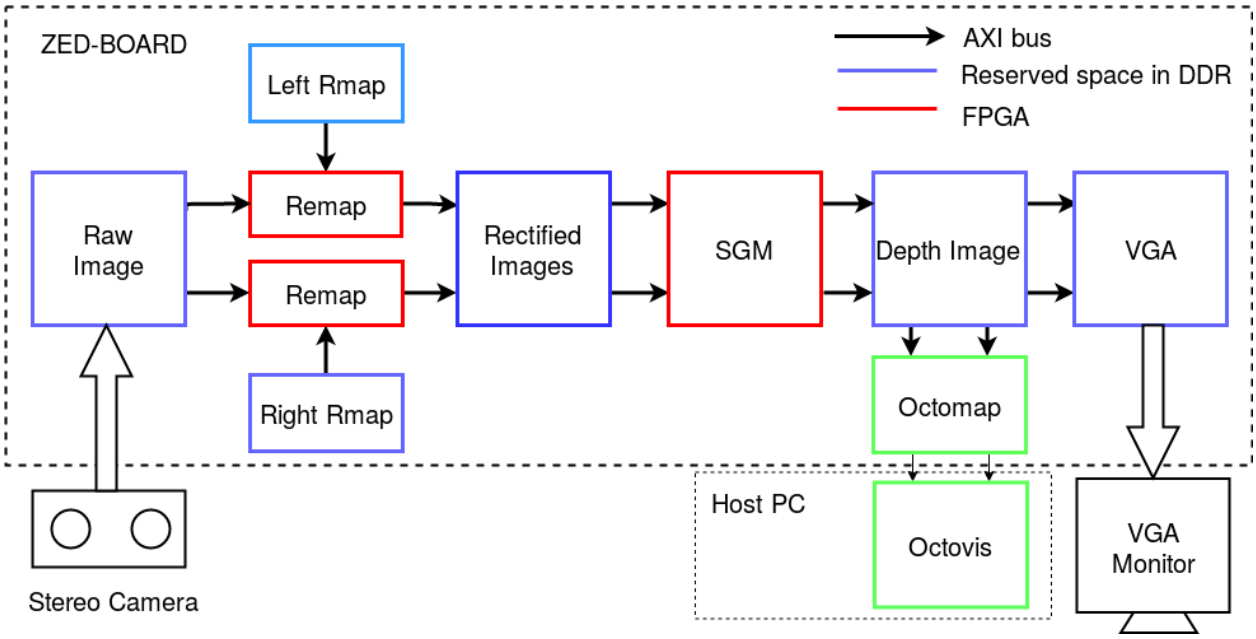


Figure 3.1: Block diagram[21]

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 the AXI4 protocol to perform memory read-write. The read-write rates are 3 orders of magnitude greater than the compute times of FPGA peripherals.

3.2 Undistortion and Rectification[22]

Stereo camera calibration and rectification (one time step) is done using the OpenCV library. Calibration and rectification process produces distortion coefficients and camera matrix. From these parameters, using the OpenCV library, two maps are generated, one for each camera. Size of a map is the same as image size. Rectified images are built by picking up pixel values from raw images as dictated by the maps. The map entry (i,j) contains a coordinate pair (x, y); and the (i, j) pixel in the rectified image gets the value of the pixel at (x, y) from the raw image. x and y values need not be integers. In such a case, linear interpolation is used to produce final pixel value. Figure 3.2 shows the remap operation with 4 neighbour bilinear interpolation.

On-chip memory is limited in size, and it is required by the stereo-depth hardware module. So, we store the maps generated during calibration and rectification in system DDR. The map entries are in fixed-point format with five fractional bits. Captured images are stored in DDR too. The hardware module iterates over the maps, and builds up the result (left and right) images by picking pixels from raw images. Note that, while the maps can be read in a

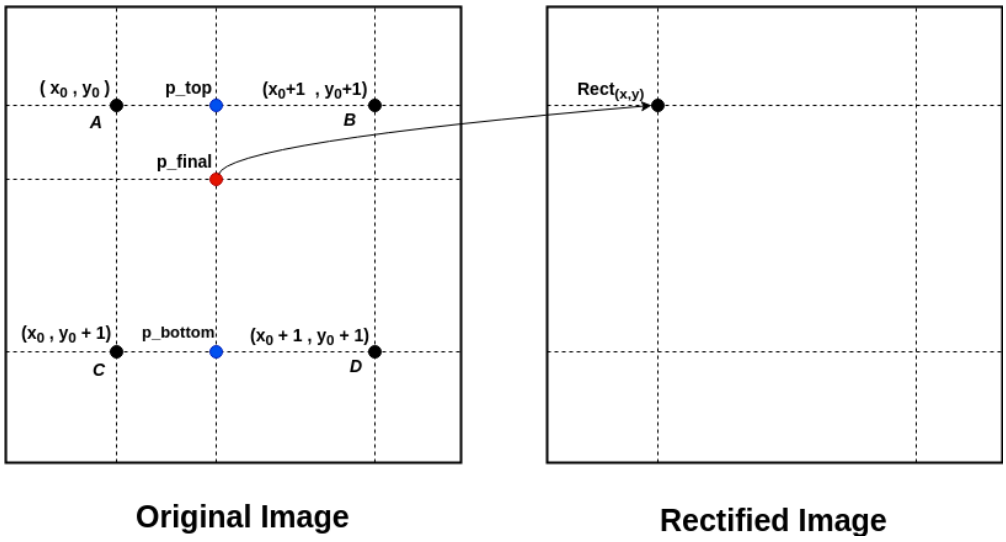


Figure 3.2: Remap operation[22]

streaming manner, the random-access is required for reading the raw images. For fractional map values, bilinear interpolation (fixed point) is performed. Resulting images are stored back in DDR. As this hardware module has to only - "read maps and raw images pixels from DDR, perform bilinear interpolation, and store the pixels back", it needs less than 5% resources of the Zynq chip.

3.3 SGM Block Architecture

In Census implementation we scan using row-major order through every pixel in the image and perform stereo matching. Thus for the SGM implementation built upon this, we consider only four neighbors for a pixel under processing as shown in red in Figure 3.3. This is done because we have the required data from neighbors along these paths. The quality degradation by using 4 paths instead of 8 paths is 2-4%[4]. Figure 3.4 shows the implemented SGM architecture. The aggregated cost for all paths and disparity indices of one row above the pixel (full row not shown in figure) and the left adjacent pixel of the current pixel are depicted as columns of colour yellow, red, blue and green for paths top left, top, top right and left respectively.We store the resultant accumulated cost which is computed using Equation 2.3. 4 Paths have been used by grouping them into single information as shown in Figure 2.1b. Thus in Equation 2.3 our n value is 4 and r has a single value for a pixel. The Census metric cost is stored in an 8bit unsigned char so the total size of memory occupied by the cost is given as $SizeofRowCostArray = (ImageWidth) * (DisparityRange) * (NoofPathGroups) = 640 * 92 * 1 = 57.5KB$.

Minimum cost across disparity search range is computed once and stored for the above row and left adjacent pixel. These scalar quantities are shown as small boxes of the same color. Since the minimum cost values are accessed multiple times, storing the minimum values instead of recomputing them every time they are required saves a lot of computations. The pixels in the row above the current pixel can be either top-left, top or top-right neighbors of the current pixels. Hence costs along the left path (green columns) are not stored for the

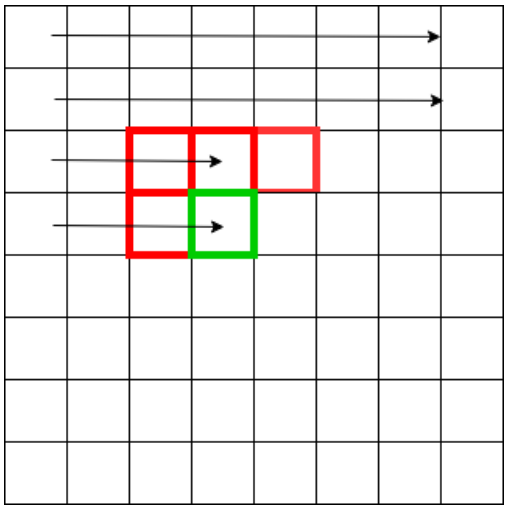


Figure 3.3: Four neighbour paths considered for SGM[21]

row above the pixel. Figure 3.4 also shows the data required and the steps for computing the aggregated cost for a certain pixel considering all the 4 paths. Smoothing term(2nd part in the RHS of Equation 2.3) along all paths are summed up to obtain a sum cost which has to be divided by n(4). Since division is resource-intensive hardware we use left a shift by 2 to divide by 4. Then the resulting value is added with the current hamming distance (1st part in the RHS of Equation 2.3). An upper bound is applied to the sum cost. The index of the minimum of this modified sum cost is the disparity for this pixel. The costs for all disparities are stored as they will be required for future pixels of the next row. The minimum cost across the disparity search range is also computed and stored for all paths.

Figure 3.5 shows the data structures used for storing the costs and the algorithm for updating them as we iterate over pixels. The *cost_row* structure has dimensions- image columns, path groups and disparity search range. It stores the costs for one row above the current pixel for all paths and disparity indices. The $cost\_left$ structure has dimensions- path groups and disparity search range. It stores the cost for the left adjacent pixel of the current pixel for all paths and disparity indices. As shown in Figure 3.5 the current pixel under processing is at row 6 column 20. It requires data from its 4 neighbors: row 5 column 19, row 5 column 20, row 5 column 21 and row 6 column 19.To generate data for current pixel we use the data of $cost\_left$ and 3 pixel vectors of *cost_row*. As we compute the disparity for this pixel and also performing the housekeeping tasks of generating the required data, we update the structures as shown in Figure 3.5. The data from $cost\_left$ is moved to the top-left neighbour of the current pixel in *cost_row*. The top left pixel cost data is not required anymore and hence is not stored. After this update is done, the currently generated data is moved into $cost\_left$.

Pixels at the top, left and right edge of the image are considered to have neighbors with a maximum value of aggregated cost. As SGM cost aggregation step is a minimization function, they are effectively ignored. The *cost_row* and $cost\_left$ structures are initialized to a maximum value before the stereo matching process. This initialization has to be done for every frame.

3.4 HLS Implementation

High-level Synthesis(HLS) platform such as Vivado HLS (from Xilinx) facilitates a suitably annotated description of compute-architecture in high level language like C or C++ , which it converts to a low-level HDL based description of the same computing architecture. The generated VHDL or Verilog code is then synthesized to target fpgas. We have used Vivado HLS tools provided by Xilinx to convert our C implementation to HDL and package it to an IP for further use. The code is attached in the Appendix. The structure of HLS stereo matching code is as follows.

```
void stereo_matching_function(){
for(int row=0; row<IMG_HEIGHT; row++) {
 for(int col=0; col<IMG_WIDTH; col++) {
  //Reading pixel from DDR through AXI4
  protocol in row-major order
  //Shifting the Census Match window in
  the left and right blocks
  for(int d=0; d<SEARCH_RANGE; d++) {
   //Match l_window with r_window[d]
   //Update the min cost index
   //Add the necessary output to the cost
   row and cost left vectors
  }
 //write disparity image pixel to DDR
 }
}
}
```

There are no operations between the row and col loop, hence they can be effectively flattened into a single loop. The plan was to pipeline the merged row-column loop. Thus resulting in increase of frame rate by disparity range times if the pipeline throughput had been 1. However the resources in fpga device on Zedboard are not enough to permit the pipelining of row column loop. Hence, only the search range loop was pipelined. The arrays used in the implementation have been partitioned effectively to reduce the latency. Based on the availability of Hardware resources we have divided the whole image into sections and disparity of each section is computed in parallel. It was observed that a frame rate of 2.1 fps is obtained with the most used resource being Block RAM (BRAM) 17%. The time required for processing one frame for such an implementation can be given as

$$T \propto no.\ of\ rows\ \times no.\ of\ columns \times (search\ range\ + pipeline\ depth\ ) \tag{3.1}$$

The characteristic of this implementation is that the logic synthesized roughly corresponds to the matching of two Census windows, the cost aggregation arithmetic and on-chip memory to store data for the next iterations. As we sequentially iterate over rows, columns and disparity search range we reuse the same hardware. Thus, the FPGA resources required are independent of the number of rows, columns and search range but computation time

required is proportional to these parameters as shown by equation 3.1. This gives us the idea to divide the images into a number of sections along the rows and process the sections independently by multiple such SGM blocks. As the most used resource is BRAM at 17%, we can fit 5 such SGM blocks with each block having to process 5 sections of the image i.e. 128 rows in parallel. Thus we increase resource usage 5 times and reduced the time required for computation by the same resulting in 10.5 fps.

One flaw to this approach is that if we divide the input image into exactly 5 parts, there will be a strip of width window size at the center of the disparity image where the pixels will be invalid. The solution to this is that the height of each section is $image\_height/5 + window\_size/2$. This is shown in Figure 3.6 for an example of 2 sections.

3.5 Hardware Setup

Figure 3.7a shows the hardware setup for zedboard. The Zed camera is connected to a USB 2.0 port of the Zedboard. The Zedboard is booted with petalinux through SD card. The only other connections to Zedboard are the connection to VGA display and power.

Figure 3.7b shows the hardware setup for Ultra96. The Zed camera is connected to a USB 3.0 port of the Ultra96. The Ultra96 is booted with PYNQ OS[18] through SD card. As the PYNQ OS has a display running, we can connect the board to a VGA monitor through the UDP port using a UDP to VGA converter or we can export the screen using ssh -XC. As the Ultrascale+ SOC comes with a GPU the display doesn't seem to have much computation trouble.

3.6 PYNQ Application Overview

PYNQ stands for python productivity for ZYNQ. This library has been developed to ease the development of an embedded system consisting of an accelerated function on a programmable logic(FPGA) specifically for Zynq and Zynq Ultrascale + devices without having to use ASIC-style design tools to design programmable logic circuits. The Hardware developer has to first develop the IP for the programmable logic. Programmable logic circuits are presented as hardware libraries called overlays. These overlays are analogous to software libraries. Using this PYNQ library we can write the bitstream to FPGA and interact with IP as an application programming interface (API). It provides the IP to a software developer as an python function. So once the overlay has been designed it can be used in any application. PYNQ provides a Python interface to allow overlays in the PL to be controlled from Python running in the PS.

A software engineer can select the overlay that best matches their application. In our case SGM computation is being done on the FPGA so whenever there is need for disparity image the python function for SGM is called with parameters being left and right images and this starts the computation on the Programmable Logic and returns the disparity matrix. An overlay usually includes:

- A bitstream to configure the FPGA fabric
- A Vivado design Tcl file to determine the available IP

- A Vivado Hardware Handoff file (.hwh) to determine the base address of the Peripherals being used.
- Python API that exposes the IPs as attributes

Figure 3.8 shows the block diagram of the python application being run on the PYNQ / Ultra96 board to process the real time images from Zedcamera in PL and display the disparity images. In Figure3.8 the blocks with blue colour use PYNQ library functions to either configure the hardware or to do the communication between PS(Processing System) and PL(Programmable Logic) and the blocks with pink colour use functions from OpenCV library. Reprint of the application Code is attached in the Appendix.

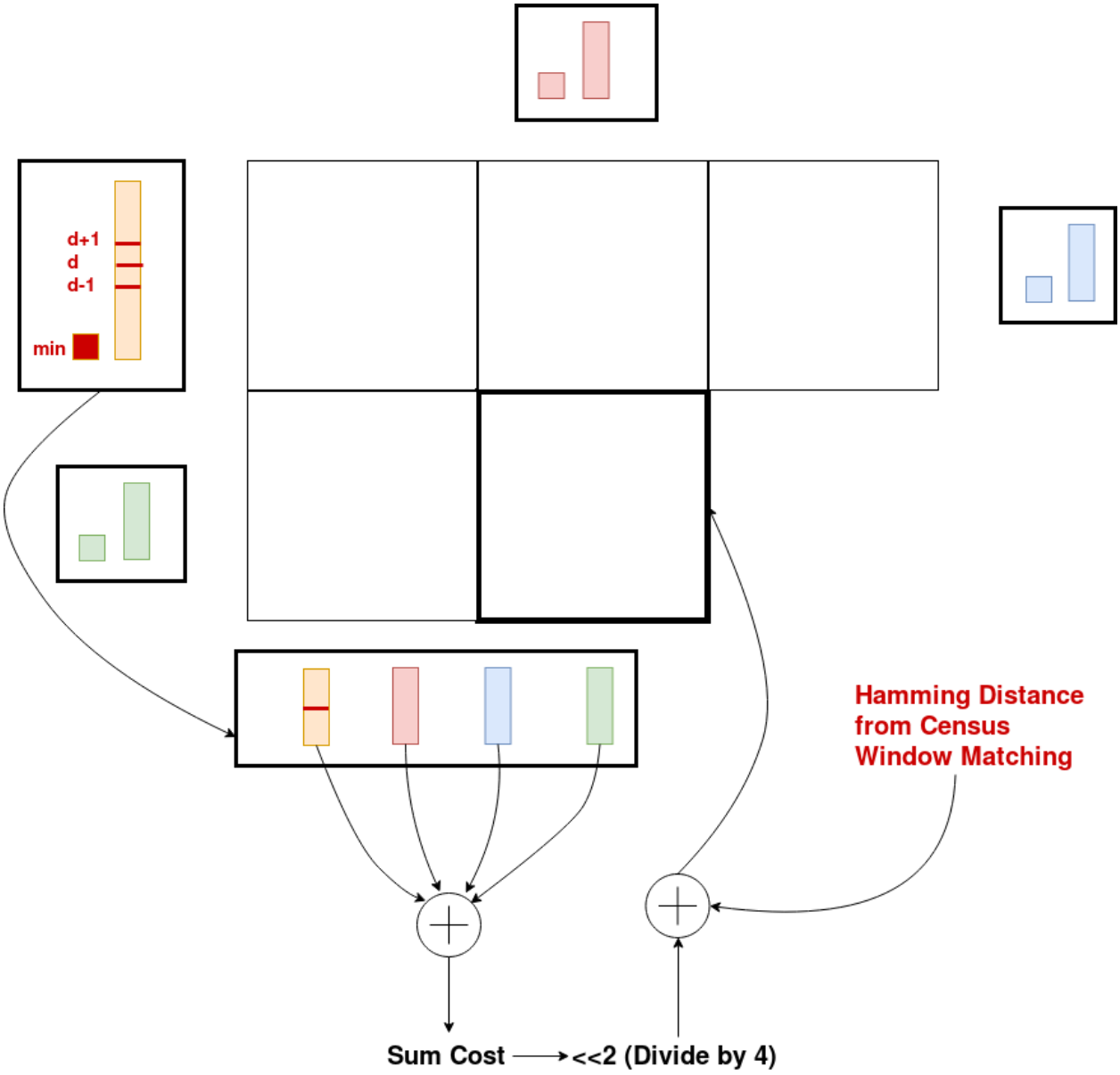


Figure 3.4: SGM Cost Computation. Steps involved in calculating the disparity for the current pixel.

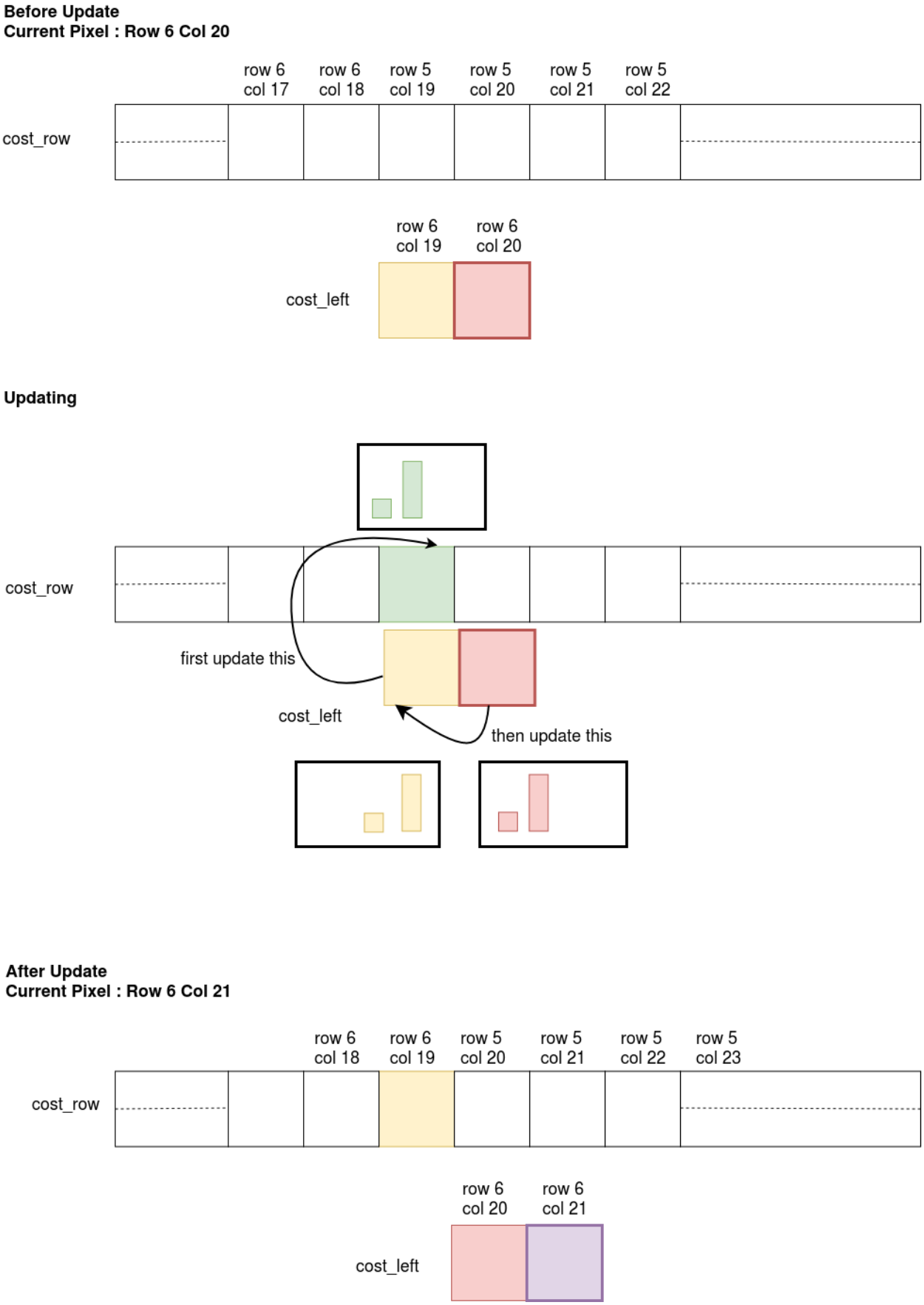


Figure 3.5: SGM Array Updation[21].

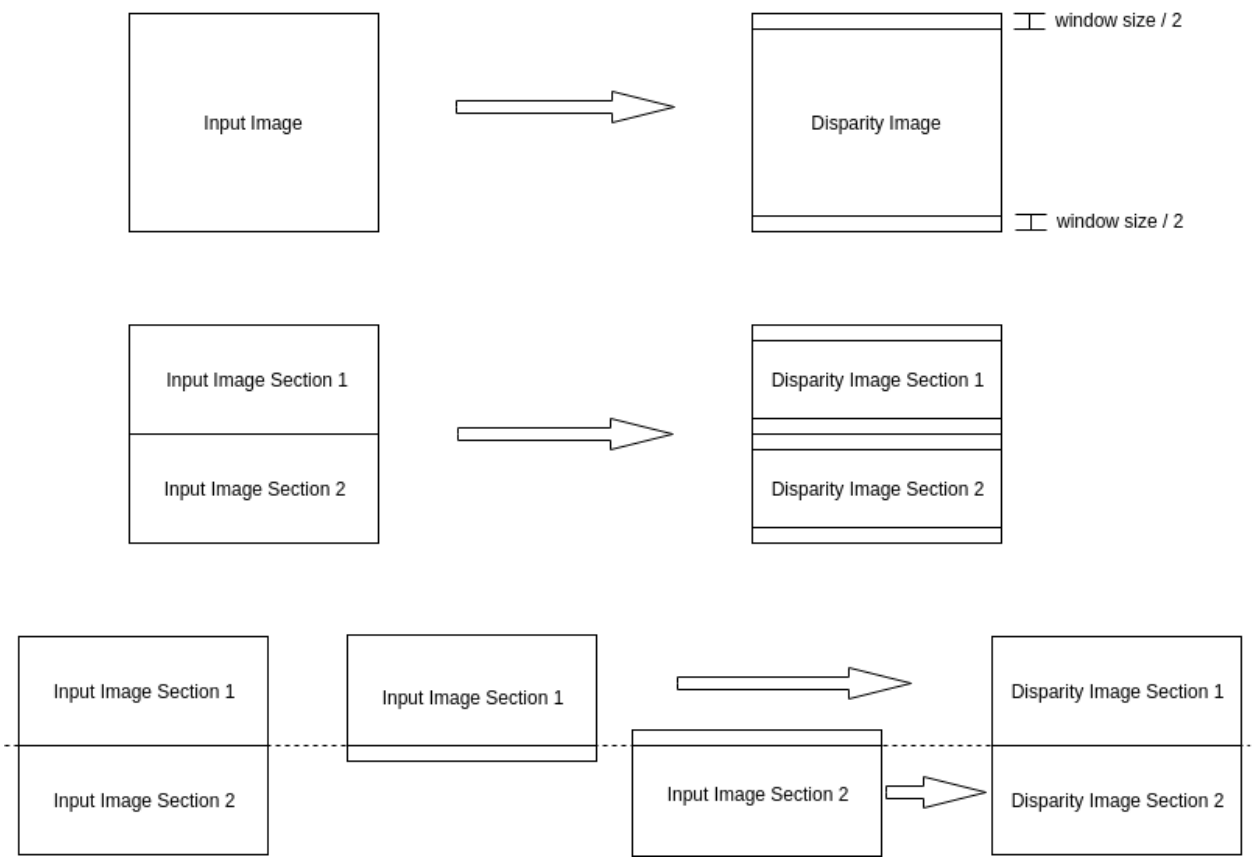


Figure 3.6: Section wise division of images for parallel processing[21]

(a) Zedboard (b) Ultra96

Figure 3.7: Hardware setup

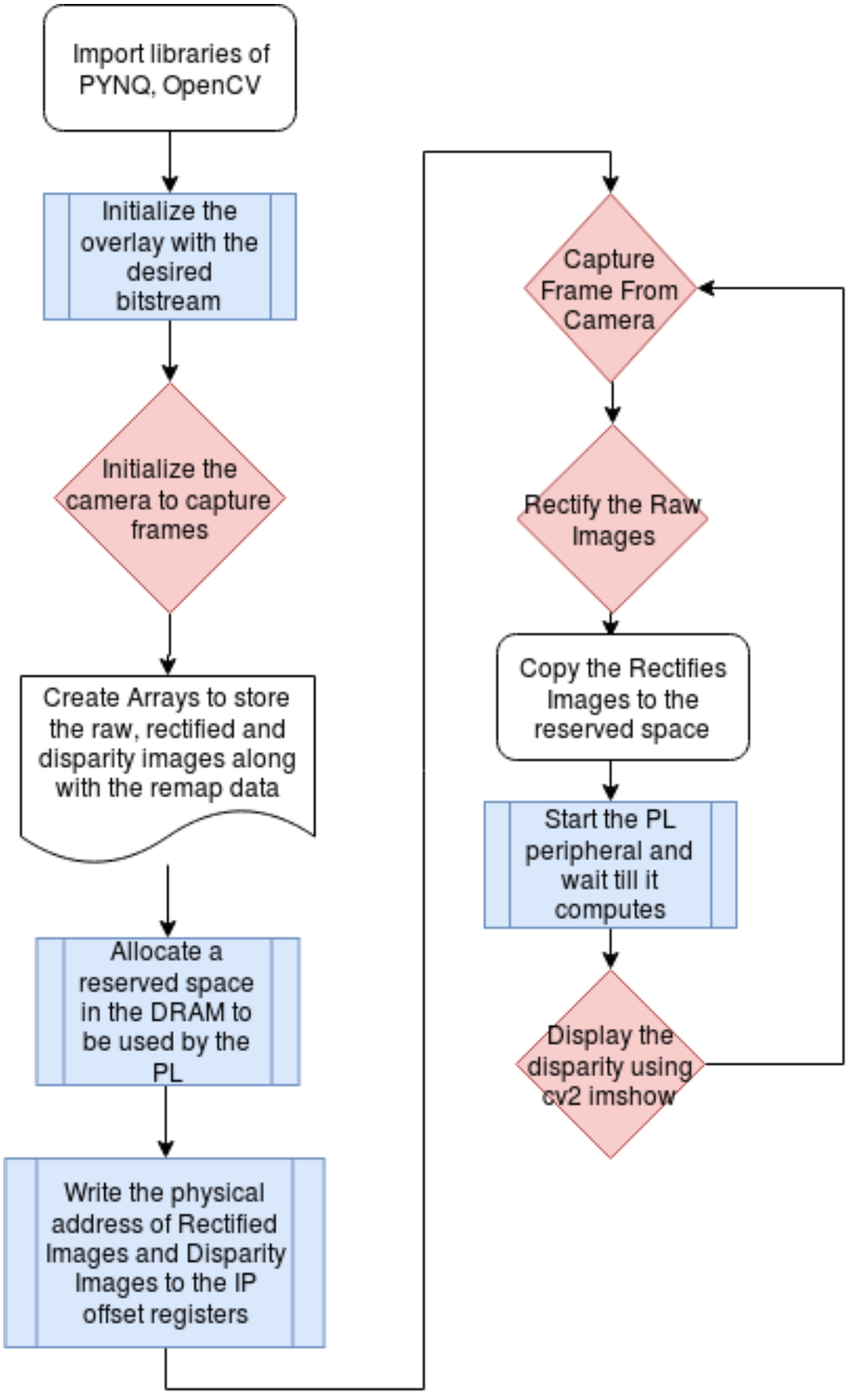


Figure 3.8: PYNQ Application block diagram

Chapter 4

Simulation and Results

4.1 Experimental Results and evaluation

The obtained frame rate for the implemented system is 10.5 fps with Zedboard running at 100 MHz. The Power consumption of the computation which is performed in FPGA is 0.72W whereas the on-chip arm processor which is being used to capture the images and start the FPGA peripherals along with the ELP stereo-camera consumes 1.68 watt , thereby raising power consumption to 2.4W. A $10m\Omega$, 1W current sense resistor is in series with the 12V input power supply on the Zedboard. Header J21 straddles this resistor to measure the voltage across this resistor for calculating Zedboard power[10]. The resource usage is summarized in Table 4.1. It is observed that the BRAM utilization is the most. This is due to storing large cost arrays.

	BRAM	DSP	FF	LUT	LUTRAM
Utilization	132	65	39159	37070	981
Available	140	220	106400	53200	17400
% Utilization	94.3	29.5	36.8	69.6	5.64

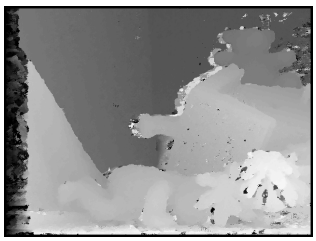
Table 4.1: Resource utilization for the entire design in Zedboard

The algorithmic performance can be measured by percentages of erroneous disparities with respect to ground truths on the Middlebury test images. A 5 pixel tolerance is considered due to intensity variation caused by changing resolution of raw image. The percentage of erroneous disparities for different images is summarized in Table 4.2. It is notable that no post processing has been done on the SGM output.

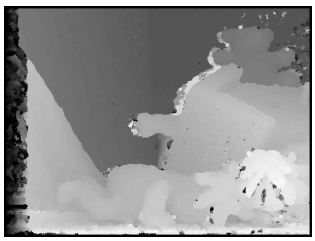
Figure 4.1 shows the software and hardware implementation results on Teddy image from Middlebury 2003 dataset[5]. Figure 4.1c-d show the results of an inhouse software implementation of SGM and Figure 4.1e shows the result of the hardware implementation. It can be observed that SGM with 8 paths gives 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 4.1f shows the SGM disparity image with $cost\_row$ and $cost\_left$ initialized to zero. Since the cost aggregation function is minimization function, the zeros

(a) Left image

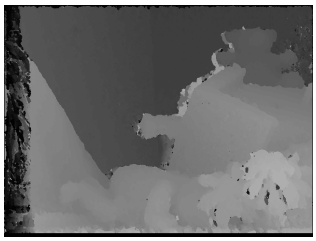
(b) Ground truth



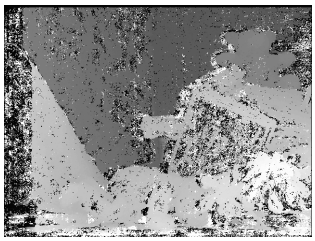
(c) SGM 4 paths software



(d) SGM 8 paths software



(e) SGM 4 paths hardware



(f) SGM with arrays initialized to zeros

Figure 4.1: SGM results on Middlebury images

from the arrays propagate to further pixels. The trickle down effect causes the degradation of the disparity image. Similar results with frame rate around 8.3 fps were also achieved by an inhouse GPU implementation of SGM on Jetson TK1 board which is of MAXWELL architecture with 256 cores and power consumption < 10 watts. This implementation is analyzed and optimized by using OpenMP for multi-threading and AVX (Advanced Vector Extension) registers for vectorization. GPU shared memory is used to reduce the global memory access. CUDA shuffle instructions are used to speed-up the algorithm and vector processing is also applied.

Fig 4.3 shows the captured image and the corresponding disparity image obtained using the SGM implementation.

Figure 4.2 shows the qualitative comparison or our results with Middlebury data set. We can see that the objects placed near are not accurate this is because we have used the disparity range of 92 pixels and so it is not able to find a match in the corresponding left and right images. Thus for a better accuracy, disparity range can be increased with the trade-off being update rate as the pipeline latency will increase. Table 4.3 shows the comparison of hardware utilization between our approach and [6] which shows this implementation uses much lesser Hardware Resources and thus having less power consumption. Furthermore, if we were to use fpga used in [6], we would have far more liberty with resources that can be

Figure 4.2: Qualitative Comparison of our results with some of the Middlebury data set. 1st Row contains the Left Raw Images, 2nd Row contains the ground truth of the corresponding Images and 3rd Row contains the Output of our Implementation.

Image	SGM
Teddy	11
Dolls	17
Books	20
Moebius	20
Laundry	27
Reindeer	27
Art	30

Table 4.2: Percentage error of disparity image pixels as compared to ground truth for Middlebury images

(a) Left image classroom

(b) SGM disparity image classroom

Figure 4.3: SGM results on ZED camera image

leveraged to further pipeline the design and obtain another order of speedup. However we have focused on very low power consumption as well as small form factor that is necessary for drones vision, blind aid etc. We compare the results of our implementation in different FPGAs in the next section.

Figure 4.4 shows the corresponding disparity image using an exported display on a VGA monitor. The camera can be seen on the left side of the image.

	BRAM18K	DSP	FF	LUT	Frame Rate	Power (Approx)
Ours	132	65	39159	37070	10.5	0.72W
[6]	163	-	153000	109300	72	3W

Table 4.3: Comparison of FPGA Hardware Resources(Approx) and power consumption between our approach and [6]

4.2 Hardware Utilization and Update Rate comparison across FPGA's

The Pseudo-code for the SGM IP synthesised using Vivado HLS is as follows with loops labelled for a better understanding of the HLS results.

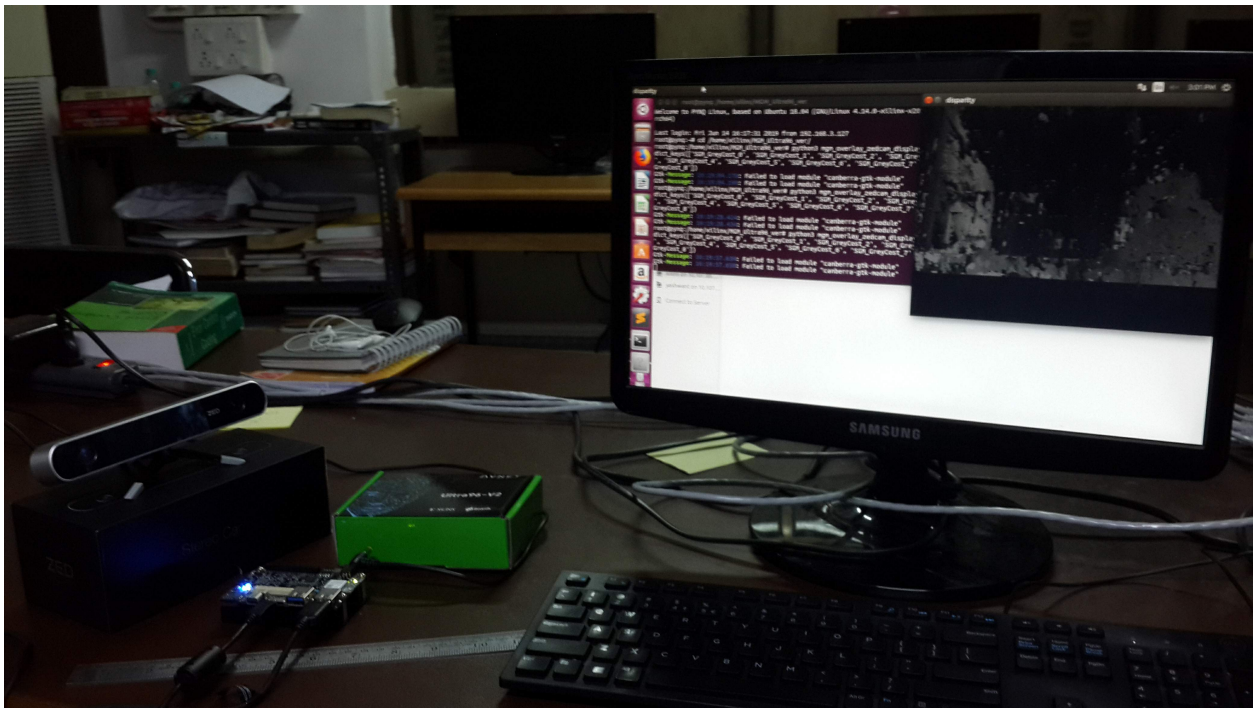


Figure 4.4: Scene and disparity image on exported VGA monitor

```
void stereo_matching_function(){
 init1,2://Initializing the cost matrices to a max value.
 L1:for(int row=0; row<IMG_HEIGHT; row++) {
  L2:for(int col=0; col<IMG_WIDTH; col++) {
   //Reading pixel from DDR through AXI4 protocol in row-major order
   //Shifting the Census Match window in the left and right blocks
   sgm_compute:for(int d=0; d<SEARCH_RANGE; d++) {
    //Match l_window with r_window[d]
    //Update the min cost index
    //Add the necessary output to the cost row and cost left vectors
   }
  //write disparity image pixel to DDR
  }
 }
}
```

The loops L1 and L2 have been flattened into a single loop L1_L2 as there is no logic in between them. We now have a scope to pipeline 2 loops:

- Loop L1_L2 with range of the total number of pixels in the image. For 640x480 resolution we have loop size 307200.
- Loop sgm_compute with range as the total possible disparities or the search range in the image. In our case its value in 91.

Current Frame rates for the following boards Zedboard and Ultra96 are 10.5 , 16 fps respec-

tively tested on the hardware and for ZCU104 the estimated frame rate of the synthesised design is 84 fps.

4.2.1 Zedboard

Zedboard an evaluation board for Xilinx Zynq-7000 SoC. It comes with FPGA part number XC7Z020-CLG484-1 and Dual-core ARM Cortex-A9 MPCore. For the efficient utilization of the resources on zedboard only the sgm_compute loop has been pipelined, scheduling every next disparity computation for a particular pixel after 1 cycle. This can be seen in Figure 4.6a where sgm_compute initiation interval is 1. Based on the resource utilization we divided the image into 5 sections which are processed in parallel. Table 4.4 contains the resource utilization for 5 SGM IP's along with 2 ReMap IP's and a VGA peripheral. The Latency of the SGM block is 9394590 cycles for a 100MHz clock(Figure 4.6a). This computes to a frame rate of **10.5 fps**.

	BRAM	DSP	FF	LUT	LUTRAM
Utilization	132	80	39159	37070	2681
Available	140	220	106400	53200	17400
% Utilization	94.3	36.36	36.8	69.6	15.51

Table 4.4: Resource utilization for the entire system on Zedboard

4.2.2 AES-ULTRA96-V2-G

Ultra96 is an ARM-based, Xilinx Zynq UltraScale+ MPSoC development board based on the Linaro 96Boards specification. It comes with FPGA part number xczu3eg-sbva484-1-e. Loop L1_L2 has been pipelined considering there are more number of LUTs available on this board. From Figure 4.6b it can be seen that the initiation interval of loop L1_L2 is 17 cycles compared to the zedboard case Figure 4.6a which has L1_L2 latency o f around 140 cycles. This shows an improvement of 8 times. It could not achieve the target pipeline initiation interval of 1 cycle because of the Limited Read-Write ports of the Block Ram. A search range of 64 has been used for this and the following board. From the execution latency of 5288791 cycles for 100MHz clock we get a frame rate of 18 fps. From the resource utilization Table 4.5 of ultra96 we can see that maximum utilized resource is LUT which is 53.9%. So Unlike for the zedboard where we divided the image into 5 sections here we could not do that. While this design is tested on board the result was not as expected 18 fps but was less. The reason could be the wrong estimation of HLS by not considering the memory read write cycles to DRAM correctly. The design explained in section 4.2.1 was also tested on the ultra96 but with 9 sections instead of just 5 as was the case on zedboard owing to more resources in the former. This gave **16 fps** on realtime input images from zedcamera. One should note that as we increase the number of sections the accuracy of the disparity image gets reduced as now the smoothness cost is propagated only to a lesser part of the image.

	BRAM	FF	LUT	LUTRAM
Utilization	73.5	14240	38030	971
Available	216	141120	70560	28800
% Utilization	34	10.09	53.9	3.37

Table 4.5: Resource utilization on Ultra96

4.2.3 ZCU104

The ZCU104 is also a Zynq UltraScale+ MPSoC device development board which comes with FPGA part number XCZU7EV-2FFVC1156 and a quad-core ARM Cortex-A53 applications processor. The HLS implementation for this board is similar to that of Ultra96 with loop L1_L2 pipelined. The difference is in this board 5 sections were able to fit. So the estimated latency of the single section has reduced to 1150784 cycles(Figure **??**) achieving a frame rate of **84 fps**.

	BRAM	FF	LUT	LUTRAM
Utilization	367.5	58393	193402	2605
Available	912	548160	274080	144000
% Utilization	40.3	10.65	70.56	1.81

Table 4.6: Resource utilization on ZCU104

4.3 3D Map visualization using Octomap

Octomap library has been used to convert the disparity image to point cloud. It has a tree data structure with each node having 8 child nodes. So the volume cube is subdivided into 8 parts and based on the depth each part is further divided into 8 sub-parts and so on. The inputs of the program being the origin of the point cloud which is the current pose of camera and the real co-ordinates of each pixel(X,Y,Z) . A program has been written to generate the real co-ordinates of each pixel(X,Y,Z) computed using the Equation 4.1,4.2,4.3 where p,q are the corresponding row and columns and the remaining parameters were obtained through calibration file and passed to the octomap library. This library has been cross compiled for ARM processor and has been tested on Zedboard as well as PYNQ board. Point cloud file is computed on board and the file is visualized using octovis program on a host computer.

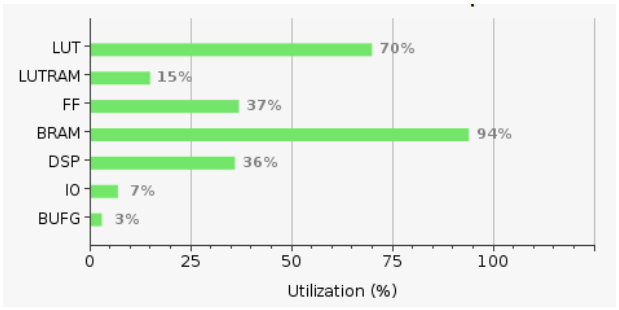
Figure 4.7 shows a computed point cloud and its visualization of an online ground truth disparity image.

Figure 4.8 show the computation of the disparity image and the resulting 3d point cloud on the zedboard visualized using octovis on a host computer.

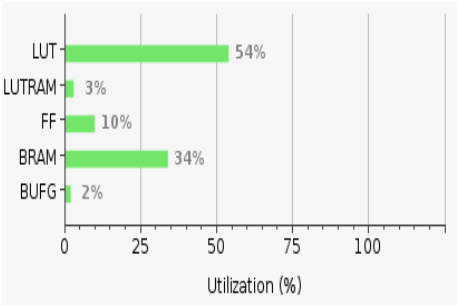
$$X(p,q) = (p - Optical\_Center\_X) * Z(p,q)/(FocalLength) \tag{4.1}$$

$$Y(p,q) = (q - Optical\_Center\_Y) * Z(p,q)/(FocalLength) \tag{4.2}$$

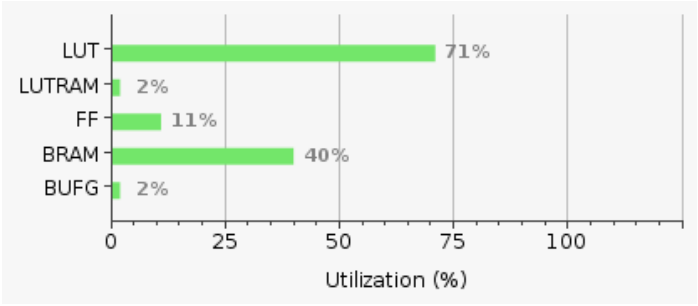
$$Z(p,q) = Depth(p,q) = Baseline * (FocalLength)/disparity \tag{4.3}$$



(a) Zedboard



(b) Ultra96



(c) ZCU104

Figure 4.5: % Utilization of the systems in different boards

4.4 Stereo Visual Odometry Application using Disparity form FPGA on PYNQ board

In order to test the disparity computation on hardware an example application of Stereo Visual odometry was choosen. Given a series of images taken by a stereo camera, stereo visual odometry is able to give the location or the pose of the camera for the corresponding image taken.

2000 Stereo Images from kitti dataset were used on **PYNQ board** with disparity image being generated on the programmable logic and the pose estimation being computed on ARM processing system by a code implemented in python. The reference project was taken from [20]. This project uses opencv disparity generation function but we need to use the disparity using our custom FPGA solution. This has been implemented using PYNQ overlay methodology and the updated code is in the following github link. Main file is in src/SVO.py . Couple of images which were used are attached in the Figure 4.9 for reference.

https://github.com/temburuyk/Stereo-visual-odometry.git

Product family: zynq
Target device: xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
8947990	9394590	8947991	9394591	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- init1	79361	79361	2560	-	-	31	no
+ init1.1	852	852	4	-	-	213	no
+ init1.2	852	852	4	-	-	213	no
+ init1.3	852	852	4	-	-	213	no
- init2	426	426	2	-	-	213	no
- L1_L2	8868200	9314800	139 ~ 146	-	-	63800	no
+ sgm_compute	94	94	7	1	1	89	yes

(a) Zedboard

Product family: zynquplus
Target device: xczu3eg-sbva484-1-e

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5288790	5288790	5288791	5288791	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- init1	82048	82048	1282	-	-	64	no
+ init1.1	1280	1280	2	-	-	640	no
- init2	640	640	1	-	-	640	no
- L1_L2	5206097	5206097	35	17	1	306240	yes

(b) Ultra96

Product family: zynquplus
Target device: xczu9eg-ffvb1156-2-i

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1150784	1150784	1150785	1150785	none

Detail

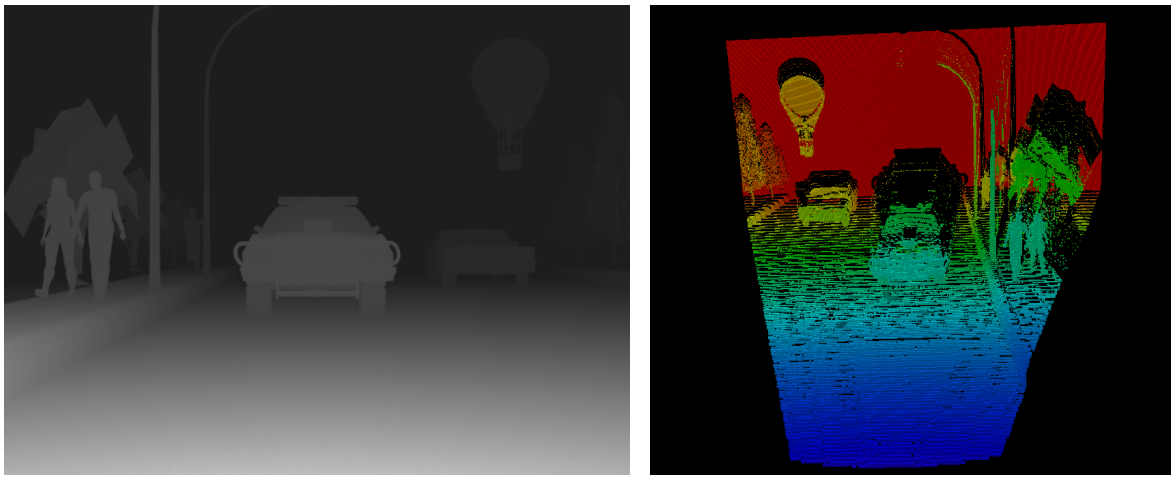
Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- init1	65737	65737	2988	-	-	22	no
+ init1.1	852	852	4	-	-	213	no
+ init1.2	1066	1066	5	-	-	213	no
+ init1.3	1066	1066	5	-	-	213	no
- init2	426	426	2	-	-	213	no
- L1_L2	1084616	1084616	34	17	1	63800	yes

(c) ZCU104

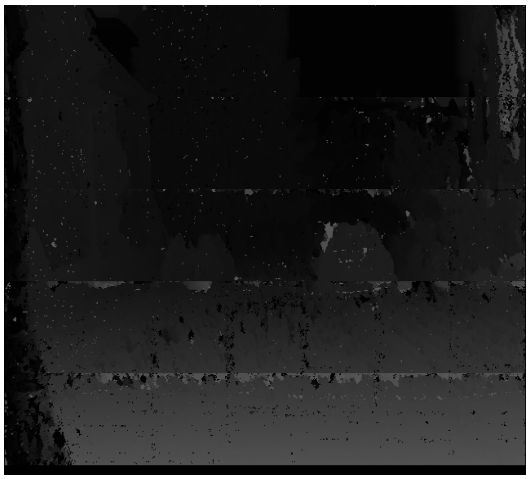
Figure 4.6: HLS performance estimates for different boards



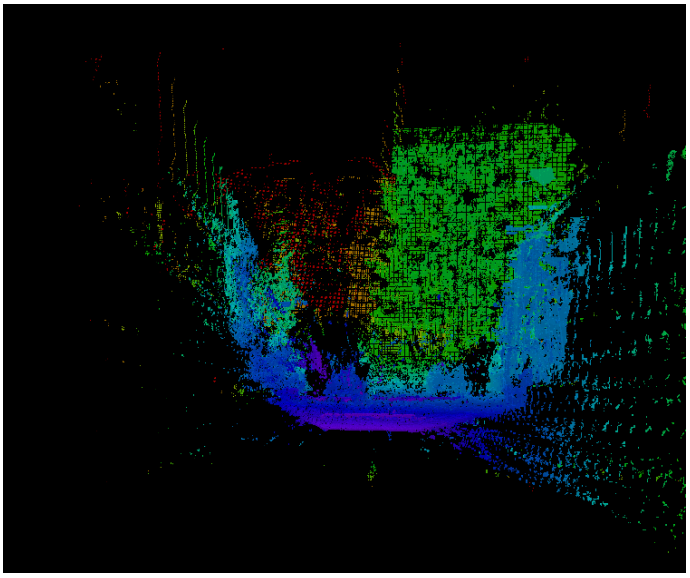
(a) Online Ground Truth disparity Image (b) 3D View(Lateral Inversion

Figure 4.7: Octree Visualization of a ground truth disparity

(a) Raw Left Image



(b) Disparity Image computed on zedboard



(c) 3D Map Visualization (Lateral Inversion)

Figure 4.8: Complete 3D view of an example image using octree

Figure 4.9: Couple of images from Data set for Odometry(taking right turn)

Figure 4.10: Path generated by the Stereo Visual Odometry for 2000 images

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The hardware implementation of the MGM[7] which is a variant of SGM[3] on Zedboard[10] / PYNQ / Ultra96 an FPGA-ARM based SOC inspired by R3SGM[6] has been presented. In order to reduce the memory consumption, we have grouped 4 paths- left, top left, top, and top right, whose pixel data are available while processing as a result of row-major order streaming process. The efficient utilization of hardware resources resulted in a low power consumption of 0.72W for data processing on FPGA that computes the Rectification and disparity Map generation and with 1.68W for data acquisition from Cameras along with starting the peripherals using the on board ARM processor achieving an update rate of 10.5Hz with a good accuracy as was shown in Table4.2 and Figure4.2. This system is highly suitable to be used in micro UAVs, blind Aids or any portable types of equipment with a small form factor and high power constraints.

5.2 Future Work

- Implementation of Iterative Closet Point(ICP) Algorithm is to be done in order to stitch the continuous 3D map being generated by the disparity images.
- After doing a literature review on current Stereo Visual Odometry algorithms a suitable FPGA implementation is to be implemented for a better performance which will use the disparity images computed by the current system.
- Try to incorporate SLAM after completing the above tasks.

Chapter 6

Appendix

6.1 PYNQ Application code

```
from pynq import Overlay
from pynq import Xlnk
import numpy as np
import cv2
import cffi

IMG_WIDTH          =640
IMG_HEIGHT         =480
SECTIONS         =10
SECTION_HEIGHT   =(int)((IMG_HEIGHT-2*FILTER_OFFS)/SECTIONS)
DISP_IMG_HEIGHT =SECTIONS*SECTION_HEIGHT
BYTES_PER_PIXEL =1
TOTAL_BYTES      =DISP_IMG_HEIGHT*IMG_WIDTH*BYTES_PER_PIXEL
ADDRESS_OFFSET   =int(TOTAL_BYTES/SECTIONS)
image_size = int(IMG_WIDTH*IMG_HEIGHT)
ZED_IMAGE_WIDTH    =1344
ZED_IMAGE_WIDTH_2 = int(ZED_IMAGE_WIDTH/2)
ZED_IMAGE_HEIGHT   =376

# This will write the bitstream to the FPGA
overlay = Overlay('path_to_.bit_file')

#Being used to convert FPGA accessable memory to np array
ffi = cffi.FFI()

#Initialize the frame capture of zedcamera connected to USB2.0
capl = cv2.VideoCapture(0)

#Initializing the arrays to be used for Image frames
imagel = np.zeros((ZED_IMAGE_HEIGHT,ZED_IMAGE_WIDTH),dtype=np.ubyte)
```

```
imager = np.zeros((ZED_IMAGE_HEIGHT,ZED_IMAGE_WIDTH),dtype=np.ubyte)
rectified_left = np.zeros((ZED_IMAGE_HEIGHT,ZED_IMAGE_WIDTH),dtype=np.ubyte)
rectified_right = np.zeros((ZED_IMAGE_HEIGHT,ZED_IMAGE_WIDTH),dtype=np.ubyte)
buffer_left = np.zeros((IMG_HEIGHT,IMG_WIDTH),dtype=np.ubyte)
buffer_right = np.zeros((IMG_HEIGHT,IMG_WIDTH),dtype=np.ubyte)

fs_left = cv2.FileStorage(\
        "left_cam.yml_file", cv2.FILE_STORAGE_READ)
fs_right = cv2.FileStorage(\
        "right_cam.yml_file", cv2.FILE_STORAGE_READ)
left_rmap0 = fs_left.getNode("rmap0").mat()
left_rmap1 = fs_left.getNode("rmap1").mat()
right_rmap0 = fs_right.getNode("rmap0").mat()
right_rmap1 = fs_right.getNode("rmap1").mat()

#Prints all the IP names in the PL bitstream
print(overlay.ip_dict.keys())
SGM_GreyCost_0 = overlay.SGM_GreyCost_0

#sgm offset registers
inL_offs = SGM_GreyCost_0.register_map.inL.address
inR_offs = SGM_GreyCost_0.register_map.inR.address
outD_offs = SGM_GreyCost_0.register_map.outD.address
CTRL_reg_offset = SGM_GreyCost_0.register_map.CTRL.address

#Object to be created to use a reserved memory space on DRAM for PL
xlnk = Xlnk()

BufferSize = 0x0500000
Image_buf =  xlnk.cma_alloc(BufferSize, data_type = "unsigned char")

#converts c data object to python memory readable object
Img_py_buffer = ffi.buffer(Image_buf,BufferSize)
Image_buf_phy_addr = xlnk.cma_get_phy_addr(Image_buf)

#Mapping the left and right images to the buffer space
LeftImg = np.frombuffer(Img_py_buffer, dtype=np.ubyte, \
        count = image_size,offset = 0*image_size).reshape((IMG_HEIGHT,IMG_WIDTH))
RightImg = np.frombuffer(Img_py_buffer, dtype=np.ubyte, \
        count = image_size,offset = 1*image_size).reshape((IMG_HEIGHT,IMG_WIDTH))
#Mapping the dispartiy images to the buffer space
disp_im_buffer = np.frombuffer(Img_py_buffer, dtype=np.ubyte, \
        count = image_size,offset = 2*image_size).reshape((IMG_HEIGHT,IMG_WIDTH))

#Writing the pointer locations of the input images to the IP offset registers
```

```
SGM_GreyCost_0.write(inL_offs,Image_buf_phy_addr+0*image_size)
SGM_GreyCost_0.write(inR_offs,Image_buf_phy_addr+1*image_size)
SGM_GreyCost_0.write(outD_offs,Image_buf_phy_addr+2*image_size)

#Continously capturing images and sending the
#frames for computation of disparity in PL
count = 0
while(count < 5000):
        #print(count)
        count = count +1
        ret, framel = capl.read()
        image = cv2.cvtColor(framel, cv2.COLOR_BGR2GRAY)
        imagel[0:ZED_IMAGE_HEIGHT,0:ZED_IMAGE_WIDTH_2] \
                = image[0:ZED_IMAGE_HEIGHT,0:ZED_IMAGE_WIDTH_2]
        imager[0:ZED_IMAGE_HEIGHT,0:ZED_IMAGE_WIDTH_2] \
                = image[0:ZED_IMAGE_HEIGHT,ZED_IMAGE_WIDTH_2:ZED_IMAGE_WIDTH]
        rectified_left = cv2.remap (imagel, left_rmap0,\
                left_rmap1, cv2.INTER_LINEAR)
        rectified_right = cv2.remap (imager, right_rmap0,\
                right_rmap1, cv2.INTER_LINEAR)
        buffer_left[0:ZED_IMAGE_HEIGHT,0:640] \
                = rectified_left[0:ZED_IMAGE_HEIGHT,0:IMG_WIDTH]
        buffer_right[0:ZED_IMAGE_HEIGHT,0:640] \
                = rectified_right[0:ZED_IMAGE_HEIGHT,0:IMG_WIDTH]
        np.copyto(LeftImg,buffer_left)
        np.copyto(RightImg,buffer_right)
        #Starting the peripheral
        SGM_GreyCost_8.write(CTRL_reg_offset,0b00000001)
        cv2.imshow('disparity',disp_im_buffer)
        cv2.waitKey(1)

#Freeing up reserved memory space and turnig off the camera feed
capl.release()
xlnk.cma_free(Image_buf)
```

6.2 SGM HLS code

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <stdio.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>
#include "ap_int.h"
```

```
using namespace std;

#define FILTER_SIZE     7
#define FILTER_OFFS     int(FILTER_SIZE/2) //4

#define SECTIONS        5
#define IP_IMG_HEIGHT   480
#define IMG_HEIGHT      (int((IP_IMG_HEIGHT-2*FILTER_OFFS)/SECTIONS)
                        + 2*FILTER_OFFS);
//480-2*4 = 472 is the full op img height
//As it is divisible by SECTIONS, it can be divided into
//integer no. of sections.
//To avoid top-bottom discontinuities we read FILTER_OFFS
//no of rows at top and bottom
#define IMG_WIDTH       640
#define SEARCH_RANGE    IMG_WIDTH/7 //Searching 1/10 of image width
#define X               (FILTER_SIZE)*(FILTER_SIZE)
#define SMALL_PENALTY   2
#define LARGE_PENALTY   20
#define PATHS           4
#define MAX_COST        127                             // 2**7 - 1
#define MAX_SUM         1023                            // 2**10 - 1

typedef ap_uint<8> cost_t;
typedef ap_uint<8> sum_t;
typedef unsigned char pix_t;


cost_t census_incremental(pix_t windowL[FILTER_SIZE][FILTER_SIZE],
    pix_t windowR[FILTER_SIZE][FILTER_SIZE]);
sum_t cost_along_path(sum_t mfd_cost_d, sum_t mfd_cost_dminus,
    sum_t mfd_cost_dplus, sum_t mfd_cost_min);
sum_t min(sum_t var1, sum_t var2, sum_t var3, sum_t var4);

void SGM_GreyCost(pix_t* inL,pix_t* inR,pix_t* outD)
{

    #pragma HLS INTERFACE m_axi port=inL offset=slave
    #pragma HLS INTERFACE m_axi port=inR offset=slave
    #pragma HLS INTERFACE m_axi port=outD offset=slave
    #pragma HLS INTERFACE s_axilite port=return

    cost_t ham_dist[SEARCH_RANGE];

    short int row, col;
```

```
    pix_t disparity;
    pix_t windowL[FILTER_SIZE][FILTER_SIZE];
    // sliding window
    #pragma HLS ARRAY_PARTITION variable=windowL complete dim=0
    pix_t windowR[FILTER_SIZE][FILTER_SIZE];
    // sliding window
    #pragma HLS ARRAY_PARTITION variable=windowR complete dim=0

    pix_t right_colL[FILTER_SIZE];
    // right-most, incoming column
    pix_t right_colR[FILTER_SIZE];
    // right-most, incoming column

    static pix_t line_bufferL[FILTER_SIZE][IMG_WIDTH];
    // line-buffers
    #pragma HLS ARRAY_PARTITION variable=line_bufferL complete dim=1
    static pix_t line_bufferR[FILTER_SIZE][IMG_WIDTH];
    // line-buffers
    #pragma HLS ARRAY_PARTITION variable=line_bufferR complete dim=1

    pix_t windowR_shift[FILTER_SIZE][FILTER_SIZE];
    // sliding window
    #pragma HLS ARRAY_PARTITION variable=windowR_shift complete dim=0


//******************** DEFINED FOR SGM ********************************//
    #pragma HLS ARRAY_PARTITION variable=cost_row cyclic factor=1 dim=1

    sum_t min_cost_row[IMG_WIDTH];
    //#pragma HLS ARRAY_PARTITION variable=min_cost_row cyclic factor=3

    //initialize cost_row to maximum 7 bit number
    init1:for(int i=0; i<SEARCH_RANGE; i++)
    {
        #pragma HLS unroll factor=3
        for(int j=0; j<IMG_WIDTH; j++)
        {
            #pragma HLS unroll factor=3
            cost_row[j][i] = MAX_SUM;
        }
    }

    //initialize min_cost_row
    init2:for(int j=0; j<IMG_WIDTH; j++)
    {
```

```
    #pragma HLS unroll factor=3
    min_cost_row[j] = MAX_SUM;
}

sum_t cost_left[SEARCH_RANGE];
//stores modified cost for left adjacent pixel
#pragma HLS ARRAY_PARTITION variable=cost_left complete
//#pragma HLS RESOURCE variable=cost_left core=RAM_2P_BRAM

sum_t min_cost_left;
//stores min cost for left adjacent pixel

sum_t cost[4],test1=0 ;
sum_t min_cost,path_cost;
#pragma HLS ARRAY_PARTITION variable=cost complete dim=1

sum_t sum_cost;
sum_t min_sum_cost;
sum_t buffer_col_1_cost[SEARCH_RANGE];
#pragma HLS RESOURCE variable=buffer_col_1_cost core=RAM_2P_BRAM
#pragma HLS ARRAY_PARTITION variable=buffer_col_1_cost complete
//initialize cost_left
for(int i=0; i<SEARCH_RANGE; i++)
{
    #pragma HLS unroll
    cost_left[i] = MAX_SUM;
}

min_cost_left = MAX_SUM;
min_cost = MAX_SUM;

L1: for(row = 0; row < IMG_HEIGHT; row++)
{
    L2: for(col = 1; col < IMG_WIDTH-1; col++)
    {
        //printf("row %d col %d\n",row,col );
        #pragma HLS loop_flatten
        //#pragma HLS pipeline II=1
        //initialize all to maximum, as min is computed on the fly
        min_sum_cost = MAX_SUM;

        pix_t winning_disp=1;

        for(unsigned char ii = 0; ii < FILTER_SIZE-1; ii++)
        {
```

```
        #pragma HLS unroll
        right_colL[ii]=line_bufferL[ii][col]=line_bufferL[ii+1][col];
        right_colR[ii]=line_bufferR[ii][col]=line_bufferR[ii+1][col];
    }

    pix_t pix_l = inL[row*IMG_WIDTH+(col)];
    right_colL[FILTER_SIZE-1] = line_bufferL[FILTER_SIZE-1][col] = pix_l;

    pix_t pix_r = inR[row*IMG_WIDTH+(col)];
    right_colR[FILTER_SIZE-1] = line_bufferR[FILTER_SIZE-1][col] = pix_r;

    //Shift from left to right the sliding window to make room for the new
    for(unsigned char ii = 0; ii < FILTER_SIZE; ii++)
    {
        #pragma HLS unroll
        for(unsigned char jj = 0; jj < FILTER_SIZE-1; jj++)
        {
            #pragma HLS unroll
            windowL[ii][jj] = windowL[ii][jj+1];
            windowR[ii][jj] = windowR[ii][jj+1];
        }
    }
    for(unsigned char ii = 0; ii < FILTER_SIZE; ii++)
    {
        #pragma HLS unroll
        windowL[ii][FILTER_SIZE-1] = right_colL[ii];
        windowR[ii][FILTER_SIZE-1] = right_colR[ii];
    }

    //Taking a copy of right window into local window
    for(unsigned char ii=0;ii<FILTER_SIZE;ii++)
    {
        #pragma HLS unroll
        for(unsigned char jj=0;jj<FILTER_SIZE;jj++)
        {
            #pragma HLS unroll
            windowR_shift[ii][jj]=windowR[ii][jj];
        }
    }
    sum_t top_left_d_prev,top_left_d,top_left_d_forw;
    sum_t top_d_prev,top_d,top_d_forw;
    sum_t top_right_d_prev,top_right_d,top_right_d_forw;
    sum_t left_d_prev,left_d,left_d_forw;

    top_left_d_prev = cost_row[0][col-1];
```

```
top_left_d = cost_row[1][col-1];
top_right_d_prev = cost_row[0][col+1];
top_right_d = cost_row[1][col+1];
top_d_prev = cost_row[0][col];
top_d = cost_row[1][col];
left_d_prev = cost_left[0];
left_d = cost_left[1];

//sum_t buffer_col_1_cost[SEARCH_RANGE];

sgm_compute:for (int d = 1; d < SEARCH_RANGE-1; d++)
{
    cost_row[col-1][d] = left_d;
    top_left_d_forw = cost_row[col-1][d+1];
    top_right_d_forw = cost_row[col+1][d+1];
    top_d_forw = cost_row[col][d+1];
    left_d_forw = cost_left[d+1];

    #pragma HLS pipeline II=1

    ham_dist[d] = census_incremental(windowL, windowR_shift);
    //returns Hamming Distance

    cost[0] = cost_along_path(top_left_d, top_left_d_prev,
        top_right_d_forw , min_cost_row[col-1]);
    cost[1] = cost_along_path(top_d, top_d_prev, top_d_forw,
        min_cost_row[col]);
    cost[2] = cost_along_path(top_right_d, top_right_d_prev,
        top_right_d_forw, min_cost_row[col+1]);
    cost[3] = cost_along_path(left_d, left_d_prev, left_d_forw,
        min_cost_left);

    top_left_d_prev = top_left_d;
    top_left_d = top_left_d_forw;
    top_right_d_prev = top_right_d;
    top_right_d = top_right_d_forw;
    top_d_prev = top_d;
    top_d = top_d_forw;

    left_d_prev = left_d;
    left_d = left_d_forw;

    //sum up costs along all paths
    path_cost = (cost[0] + cost[1] + cost[2] + cost[3]);
```

```
            sum_cost = ham_dist[d] + (path_cost)>>2;

            //Impose upper limit on sum
            if(sum_cost > PATHS*(MAX_COST + LARGE_PENALTY))
            {
                //printf("greater cost \n");
                sum_cost = PATHS*(MAX_COST + LARGE_PENALTY);
            }

            //MGM takes the sum cost as the comparing cost
            //after updating top left, update left
            cost_left[d] =  sum_cost;

            //find minimum sum cost
            if(sum_cost < min_sum_cost)
            {
                min_sum_cost = sum_cost;
                winning_disp = d;
            }
            for(unsigned char ii = 0; ii < FILTER_SIZE; ii++)
            {
                #pragma HLS unroll
                for(unsigned char jj = FILTER_SIZE-1; jj >0; jj--)
                {
                    #pragma HLS unroll
                    windowR_shift[ii][jj] = windowR_shift[ii][jj-1];
                }
            }
            for(unsigned char ii = 0; ii < FILTER_SIZE; ii++)
            {
                #pragma HLS unroll
                windowR_shift[ii][0] = line_bufferR[ii][(col)-(d)-FILTER_SIZE];
            }

        }//loop_sad_compute
        if (row >= FILTER_OFFS && row < (IMG_HEIGHT-FILTER_OFFS))
            outD[(row-FILTER_OFFS)*IMG_WIDTH + col] = winning_disp;

        //update top left
        min_cost_row[col-1] = min_cost_left;

        //after updating top left, update left
        min_cost_left = min_sum_cost;
```

```
        }//L2
    }//L1
// cout << endl;
}//function



cost_t census_incremental(pix_t windowL[FILTER_SIZE][FILTER_SIZE],
        pix_t windowR[FILTER_SIZE][FILTER_SIZE])
{
    #pragma HLS inline
    pix_t left_center = windowL[FILTER_OFFS][FILTER_OFFS];
    pix_t right_center = windowR[FILTER_OFFS][FILTER_OFFS];
    cost_t cost = 0;

    ap_uint<X> hd,census_vectorL,census_vectorR = 0;
//  int hd=0;
    for(unsigned char i=0; i<FILTER_SIZE; i++)
    {
        #pragma HLS unroll
        for(unsigned char j=0; j<FILTER_SIZE; j++)
        {
            #pragma HLS unroll
            #pragma HLS loop_flatten

            census_vectorL[i*FILTER_SIZE + j] =
                (windowL[i][j] > left_center)?1:0;
            census_vectorR[i*FILTER_SIZE + j] =
                (windowR[i][j] > right_center)?1:0;

            hd = census_vectorL ^ census_vectorR;

        }
    }

    for(short int k=0; k<FILTER_SIZE*FILTER_SIZE; k++)
    {
        #pragma HLS unroll
        cost += hd[k];
    }

    return cost;
}

sum_t cost_along_path(sum_t mfd_cost_d, sum_t mfd_cost_dminus,
```

```
    sum_t mfd_cost_dplus, sum_t mfd_cost_min)
{
    #pragma HLS inline
    return min(mfd_cost_d, mfd_cost_dminus + SMALL_PENALTY, \
        mfd_cost_dplus + SMALL_PENALTY, mfd_cost_min + LARGE_PENALTY)
        - mfd_cost_min;
}

sum_t min(sum_t var1, sum_t var2, sum_t var3, sum_t var4)
{
    #pragma HLS inline
    cost_t val, val2, val3;

    val2 = (var1 < var2) ? var1 : var2;
    val3 = (var3 < var4) ? var3 : var4;
    val  = (val2 < val3) ? val2 : val3;

    return val;
}
```

6.3 Cross-compiling opencv for Arm

Note that this is required only for Zedboard with petalinux but for PYNQ / Ultra96 the OS image comes with pre installed Opencv Libraries.

```
$] mkdir opencv_setup
$] cd opencv_setup
$] # Download zip file from https://github.com/opencv/opencv.git
$] unzip opencv-master.zip
$] mkdir build
$] cd build
$] export PATH=$PATH:/opt/Xilinx/petalinux-v2017.1/tools/linux-i386/gcc-arm-linux-g
$] cmake -DCMAKE_TOOLCHAIN_FILE=../opencv-master/platforms/linux/arm-gnueabi.toolch
    ../opencv-master/
$] make
$] make install
$] ls install
```

We used opencv version 4.0.1, which was latest at the time of setup.
Above process generates opencv libraries in `install/lib/` directory.

6.4 Cross-compiling octomap for Arm

The below procedure is for Zedboard for running the binary in petalinux. For PYNQ / Ultra96 it has a pre installed GCC compiler so we just need to copy the copy the project folder from below mentioned link and use cmake to build.

```
$] mkdir octomap_setup
$] cd octomap_setup
$] # Download zip file from https://github.com/temburuyk/octomap.git
$] unzip octomap.zip
$] mkdir build_arm
$] mkdir -p platforms/linux
$] cp path-to-opencv-master/platforms/linux/* platforms/linux/
$] source /opt/Xilinx/petalinux-v2017.1/settings.sh
$] cd build_arm
$] cmake -DCMAKE_TOOLCHAIN_FILE=../platforms/linux/arm-gnueabi.toolchain.cmake \
-DBUILD_OCTOVIS_SUBPROJECT=OFF  -DBUILD_DYNAMICETD3D_SUBPROJECT=OFF \
-v  -DOpenCV_DIR=path-to-opencv-master/build_arm ../
$] cmake ..
$] make
$] make install
$] ls install
```

Above process generates octomap binary files in `/bin` directory. File dep_image_to_bt_single_image in bin folder converts disparity image to point cloud which is written into *.bt* file. This file can be copied to a local workstation and can be visualized using octovis. Please note that we have to export the opencv install path in the board so that octomap uses the shared libraries. Below are the command to be run on the board

```
#Export the opencv library path
$] export LD_LIBRARY_PATH=/mnt/install/lib/
#Parameters are the image file , Convention for distance whcic is
#false for a convention where higher intensity depicts near objects
$] bin/dep_image_to_bt_single_image disparity_image.png false
#In the pwd a custom.bt file will be written which has the point cloud information
```

6.5 Cross-compiling userspace application executable for a C++ program.

For PYNQ / Ultra96 compilation can be done on board. For Zedboard follow the below mentioned instructions which are to be executed on a host computer.

```
$] source /opt/Xilinx/petalinux-v2017.1/settings.sh
$] arm-linux-gnueabihf-g++ -Wall -g3 -c -fmessage-length=0 -o app.o app.cpp
$] arm-linux-gnueabihf-g++ -o app.elf app.o
```

These steps compile the application C++ file `app.cpp` into an object file `app.o`, and then creates an executable `app.elf` from it.

Bibliography

[1] R. Zabih and J. Woodfill [*Non-parametric local transforms for computing visual correspondence*] In Proc. ECCV, pages 151158, 1994

[2] T. Kanade [*Development of a video-rate stereo machine*] Proceedings of International Robotics and Systems Conference (IROS'95), Pittsburgh, Pennsylvania, Aug. 5-9, 1995, pp. 95-100.

[3] H. Hirschmuller [*Stereo Processing by Semiglobal Matching and Mutual Information*] IEEE Trans. Pattern Anal. Mach. Intell., 2008,30, (2), pp. 328341

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

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

[6] Oscar Rahnama, Tommaso Cavallari, Stuart Golodetz, Simon Walker and Philip H. S. Torr . [*R3SGM: Real-time Raster-Respecting Semi-GlobalMatching for Power-Constrained Systems*] International Conference on Field-Programmable Technology (FPT), Vietnam, 2018.

[7] G. Facciolo, C. de Franchis, and E. Meinhardt [*MGM: A Significantly More Global Matching for Stereovision.*] BMVC, 2015.

[8] Rostam Affendi Hamzah and Haidi Ibrahim [*Literature Survey on Stereo Vision Disparity Map Algorithms*]vol. 2016, Article ID 8742920, 23 pages, 2016.

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

[10] Zedboard datasheet:(2019,August 25) Retrieved from http://zedboard.org/sites/default/files/documentations/ZedBoard _HW_UG_v2_2.pdf

[11] Zynq 7000 datasheet:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq -7000-Overview.pdf

[12] Vivado HLS user guide:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[13] Vivado Synthesis user guide:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado-synthesis.pdf

[14] XSCT reference guide:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug1208-xsct-reference-guide.pdf

[15] Intel Realsense D435i Depth Camera:(2019,August 25) Retrieved from https://www.intelrealsense.com/depth-camera-d435i/

[16] Zed Camera:(2019,August 25) Retrieved from www.stereolabs.com

[17] OpenCV:(2019,August 25) Retrieved from https://opencv.org/

[18] PYNQ OS:(2019,August 25) Retrieved from http://avnet.me/ultra96-pynq-image-v2.4

[19] PYNQ OS:(2019,August 25) Retrieved from: https://pynq.readthedocs.io

[20] Stereo Visual Odometry:(2019,August 25) Retrieved from: https://github.com/cgarg92/Stereo-visual-odometry.git

[21] Prathmesh Sawant, FPGA Implementation of a Real Time Stereo Vision System, Mtech Dissertation Report.

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

Chapter 7

Publications

The following paper which is re write up of the report was submitted to National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG 2019).

Single Storage Semi-Global Matching for Real Time Depth Processing

***Abstract*—Depth-map is the key computation in computer vision and robotics. One of the most popular approach is via computation of disparity-map of images obtained from Stereo Camera. Semi Global Matching (SGM) method is a popular choice for good accuracy with reasonable computation time. To use such compute-intensive algorithms for real-time applications such as for autonomous aerial vehicles, blind Aid, etc. acceleration using GPU, FPGA is necessary. In this paper, we show the design and implementation of a stereo-vision system, which is based on FPGA-implementation of More Global Matching(MGM) [7]. MGM is a variant of SGM. We use 4 paths but store a single cumulative cost value for a corresponding pixel. Our stereo-vision prototype uses Zedboard containing an ARM-based Zynq-SoC [10], ZED-stereo-camera / ELP stereo-camera / Intel RealSense D435i, and VGA for visualization. The power consumption attributed to the custom FPGA-based acceleration of disparity map computation required for depth-map is just 0.72 watt. The update rate of the disparity map is realistic 10.5 fps.**

***Index Terms*—Semi Global Matching(SGM), More Global Matching(MGM), Field Programmable Gate Array(FPGA), System on Chip(SoC), Zedboard, Census Transform, High Level Synthesis(HLS)**

I. Introduction

Although 2D and 3D LIDARs (Light Detection and Ranging Sensors) provided accuracy, they did not succeed with the economics of power and bill of materials for portable goods. Stereo cameras cost less, but need a lot of computational processing, and this aspect is getting good attention of research community , spurring the development of FPGA and GPU based acceleration of stereo-vision related computation. The low power consumption of fpga-based solutions are attractive and crucial for high performance embedded computing too.

This paper describes our design and implementation of a real-time stereo depth estimation system with Zedboard [10] (housing ARM-SoC based FPGA) at its center. This system uses Zed stereo camera [16], Intel RealSense D435i [15] or ELP stereo-camera [**?**] for capturing images. Real-time Raster-Respecting Semi-Global Matching [6] (R3SGM) along with Census Transform are used for disparity estimation. The system takes in real-time data from the cameras and generates a depth image from it. Rectification of the images, as well as stereo matching, is implemented in the FPGA whereas capturing data from USB cameras and controlling the FPGA peripherals is done via application programs which run on the hard ARM processor on Zedboard. Development of the FPGA IP's is done using High-Level Synthesis (HLS) tools. A VGA monitor is interfaced to Zedboard to display the computed depth image in real-time.

Our approach is inspired by R3SGM [6] a hardware implementation of SGM. Table III (at the later portion of the paper) shows the comparison of hardware utilization between our approach and [6] which shows ours uses much lesser Hardware Resources and thus having less power consumption. It may be emphasized that we have focused on very low power consumption as well as small form factor that is necessary for drones vision, blind aid etc.

II. Literature Review

There has been a lot of research on the topic of disparity map generation dating back to 1980s. [8] reviews most of the works including both software and hardware implementations.

A binocular Stereo Camera estimates disparity or the difference in the position of the pixel of a corresponding location in the camera view by finding similarities in the left and right image. There have been various costs governing the extent of the similarity. Some of them are Sum of Absolute Differences(SAD), Sum of Squared Differences(SSD), Normalized Cross-Correlation and the recent Rank Transform and Census Transforms. They are window-based local approaches where the cost value of a particular window in the left image is compared to the right image window by spanning it along a horizontal axis for multiple disparity ranges. The window coordinate for which the metric cost is the least is selected which gives us the disparity for that corresponding center pixel. From the disparity, the depth value is computed by equation 1 where the baseline is the distance between the optical centers of two cameras.

$$Depth = Baseline * (FocalLength)/disparity \quad (1)$$

Local window-based approaches suffer when the matching is not reliable which mostly happens when there are very few features in the surrounding. This results in the rapid variations of the disparities. This problem is solved by global approaches which use a smoothing cost to penalize wide variations in the disparity and trying to propagate the cost across various pixels. The following are some of the global approaches.

A. SGM

SGM is a stereo disparity estimation method based on global cost function minimization. Various versions 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 neighboring disparities. Mutual

information between images, which is defined as the negative of joint entropy of the two images, is used in the paper [3] as a distance metric. Other distance metrics can also be used with a 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 7x7, disparity search range 92. The image resolution is 640x480. Sum of Absolute Differences (SAD) was also considered as a matching cost function. But it was observed that SAD implementation consumes more FPGA resources than the Census implementation with same parameters. This may be due to the fact that SAD computation is an arithmetic operation whereas Census computation is a logical operation.

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 the disparity search range. The next step is cost minimization in which the minimum of this array (minimum cost) is computed and the index of the minimum cost is assigned as disparity. In SGM, an additional step of cost aggregation is performed between cost computation and cost minimization. The aggregated cost for a particular pixel p for a disparity index d is given by equation 2.

$$\begin{aligned} L_r(p,d) = C(p.d) + min(L_r(p-r,d), \\ L_r(p-r,d-1)+P_1, \\ L_r(p-r,d+1)+P_1, \\ min_i(L_r(p-r,i)+P_2)) \\ -min_k(L_r(p-r,k)) \end{aligned} \tag{2}$$

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 2. First-term $C(p,d)$ is the pixel matching cost for disparity d. In our case, it is the Hamming distance returned by Census window matching. It is apparent 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 neighbors $L'_r(p-r,)$. P_1 and P_2 are empirically determined constants. For detailed discussion refer to [3].

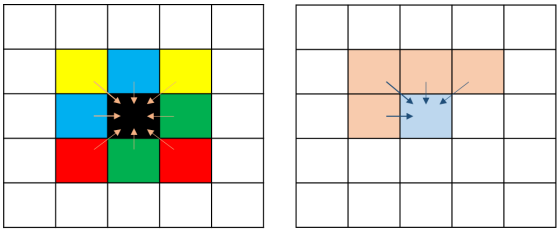
B. MGM

As SGM tries to minimize the cost along a line it suffers from streaking effect. When there is texture less surface or plane surface the matching function of census vector may return different values in two adjacent rows but due to SGM, the wrong disparity may get propagated along one of the paths and can result in streaking lines.

MGM [7] solves this problem by taking the average of the path cost along 2 or more paths incorporating information from multiple paths into a single cost. It uses this result for the next pixel in the recursion of Equation 2. The resultant aggregated cost at a pixel is then given by the Equation 3

$$\begin{aligned} L_r(p,d) = C(p.d) + 1/n \sum_{x\varepsilon\{r_n\}} (min(L_r(p-x,d), \\ L_r(p-x,d-1)+P_1, \\ L_r(p-x,d+1)+P_1, \\ min_i(L_r(p-x,i)+P_2)) \\ -min_k(L_r(p-x,k))) \end{aligned} \tag{3}$$

where n has the value depending on the number of paths that we want to integrate into the information of single cost. For example, in Figure 1a two paths are grouped into 1 so n has value 2 and there are a total of 4 groups. Thus we need to store 4 cost vectors in this case and while updating 1 cost value in the center pixel have to read cost vector of the same group from 2 pixels. Lets say $r = 1$ for blue boxes group in Figure 1a, while updating the L_r for this group of the centre pixel in Equation 3 we have x as left and top pixels. From here on SGM refers to MGM variant of it.



(a) MGM in General (b) Our Implementation

Fig. 1: Grouping of Paths in MGM

III. Hardware Architecture and Implementation

A. System Design

Figure 2 shows an overview of the implemented system. Left and right images captured from the Zed camera [16] are stored into DDR RAM (off-chip RAM). Maps required for the stereo rectification of the images are statically generated offline using OpenCV [17]. These maps are also stored into DDR RAM. We need two Remap peripherals which perform stereo rectification for the left and right images respectively. The Remap peripheral reads the raw image frame and the corresponding map and generates a rectified image frame. The rectified images are again stored into DDR. The Intel RealSense camera requires USB3.0 or higher to stream left and right images. However, Zedboard does not have USB3.0. Hence the camera cannot be directly interfaced to the board. So images were continuously captured and streamed from a computer using ethernet. The left and right image streams were received by a socket client running on the ARM processor on Zedboard. The camera outputs rectified images, hence remap peripheral is not required in this case. The images received from the socket client are stored into DDR RAM. We have also implemented it for Zed Camera [16]. For both camera modules in Binocular cameras, the stereo matching peripheral (SGM

block in the figure2) then reads the left and right rectified frame and generates disparity image which is again stored into DDR. The VGA peripheral is configured to read and display the disparity image onto a VGA monitor. FPGA peripherals perform memory access using the AXI4 protocol.

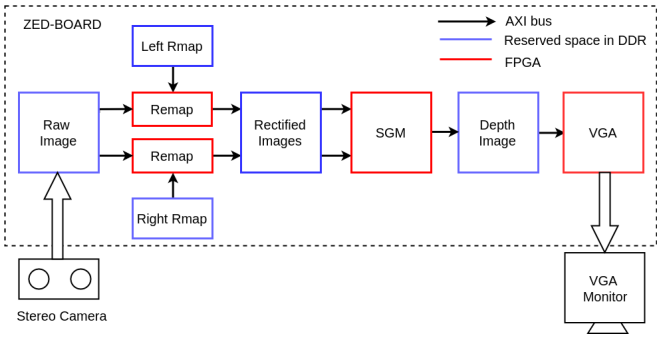


Fig. 2: Block diagram

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 the AXI4 protocol to perform memory read-write. The read-write rates are 3 orders of magnitude greater than the compute times of FPGA peripherals.

The images are captured using application programs running on the ARM processor on Zedboard. The programs make use of v4l2 library for image capture. The ARM processor is also used to control the FPGA peripherals.

B. Undistortion and Rectification

Stereo camera calibration and rectification (one time step) is done using the OpenCV library. Calibration and rectification process produces distortion coefficients and camera matrix. From these parameters, using the OpenCV library, two maps are generated, one for each camera. Size of a map is the same as image size. Rectified images are built by picking up pixel values from raw images as dictated by the maps. The map entry (i,j) contains a coordinate pair (x, y); and the (i, j) pixel in the rectified image gets the value of the pixel at (x, y) from the raw image. x and y values need not be integers. In such a case, linear interpolation is used to produce final pixel value. Figure 3 shows the remap operation with 4 neighbour bilinear interpolation.

On-chip memory is limited in size, and it is required by the stereo-depth hardware module. So, we store the maps generated during calibration and rectification in system DDR. The map entries are in fixed-point format with five fractional bits. Captured images are stored in DDR too. The hardware

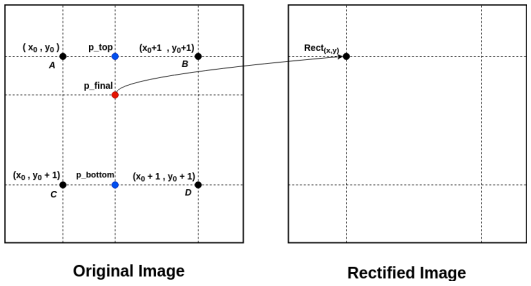


Fig. 3: Remap operation

module iterates over the maps, and builds up the result (left and right) images by picking pixels from raw images. Note that, while the maps can be read in a streaming manner, the random-access is required for reading the raw images. For fractional map values, bilinear interpolation (fixed point) is performed. Resulting images are stored back in DDR. As this hardware module has to only - "read maps and raw images pixels from DDR, perform bilinear interpolation, and store the pixels back", it needs less than 5% resources of the Zynq chip.

C. SGM Block Architecture

In Census implementation we scan using row-major order through every pixel in the image and perform stereo matching. Thus for the SGM implementation built upon this, we consider only four neighbors for a pixel under processing as shown in red in Figure 4. This is done because we have the required data from neighbors along these paths. The quality degradation by using 4 paths instead of 8 paths is 2-4% [4]. Figure 5 shows

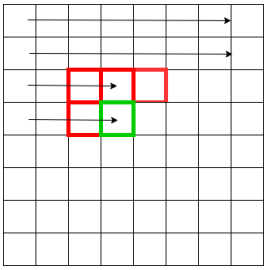


Fig. 4: Four neighbour paths considered for SGM

the implemented SGM architecture. The aggregated cost for all paths and disparity indices of one row above the pixel (full row not shown in figure) and the left adjacent pixel of the current pixel are depicted as columns of colour yellow, red, blue and green for paths top left, top, top right and left respectively.We store the resultant accumulated cost which is computed using Equation 3. 4 Paths have been used by grouping them into single information as shown in Figure 1b. Thus in Equation 3 our n value in 4 and r has a single value for a pixel. The Census metric cost is stored in an 8bit unsigned char so the total size of memory occupied by the cost is given as $SizeofRowCostArray = (ImageWidth) * (DisparityRange) * (NoofPathGroups) = 640 * 92 * 1 = 57.5KB$.

Minimum cost across disparity search range is computed once and stored for the above row and left adjacent pixel. These scalar quantities are shown as small boxes of the same color. Since the minimum cost values are accessed multiple times, storing the minimum values instead of recomputing them every time they are required saves a lot of computations. The pixels in the row above the current pixel can be either top-left, top or top-right neighbors of the current pixels. Hence costs along the left path (green columns) are not stored for the row above the pixel. Figure 5 also shows the data required

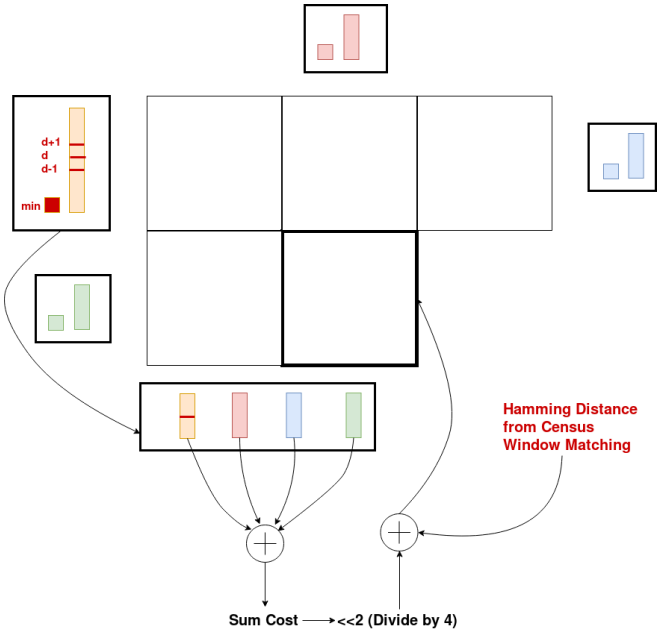


Fig. 5: SGM Cost Computation. Steps involved in calculating the disparity for the current pixel.

and the steps for computing the aggregated cost for a certain pixel considering all the 4 paths. Smoothing term(2nd part in the RHS of Equation 3) along all paths are summed up to obtain a sum cost which has to be divided by n(4). Since division is resource-intensive hardware we use left a shift by 2 to divide by 4. Then the resulting value is added with the current hamming distance (1st part in the RHS of Equation 3). An upper bound is applied to the sum cost. The index of the minimum of this modified sum cost is the disparity for this pixel. The costs for all disparities are stored as they will be required for future pixels of the next row. The minimum cost across the disparity search range is also computed and stored for all paths.

Figure 6 shows the data structures used for storing the costs and the algorithm for updating them as we iterate over pixels. The $cost\_row$ structure has dimensions- image columns, path groups and disparity search range. It stores the costs for one row above the current pixel for all paths and disparity indices. The $cost\_left$ structure has dimensions- path groups and disparity search range. It stores the cost for the left adjacent pixel of the current pixel for all paths and disparity indices.

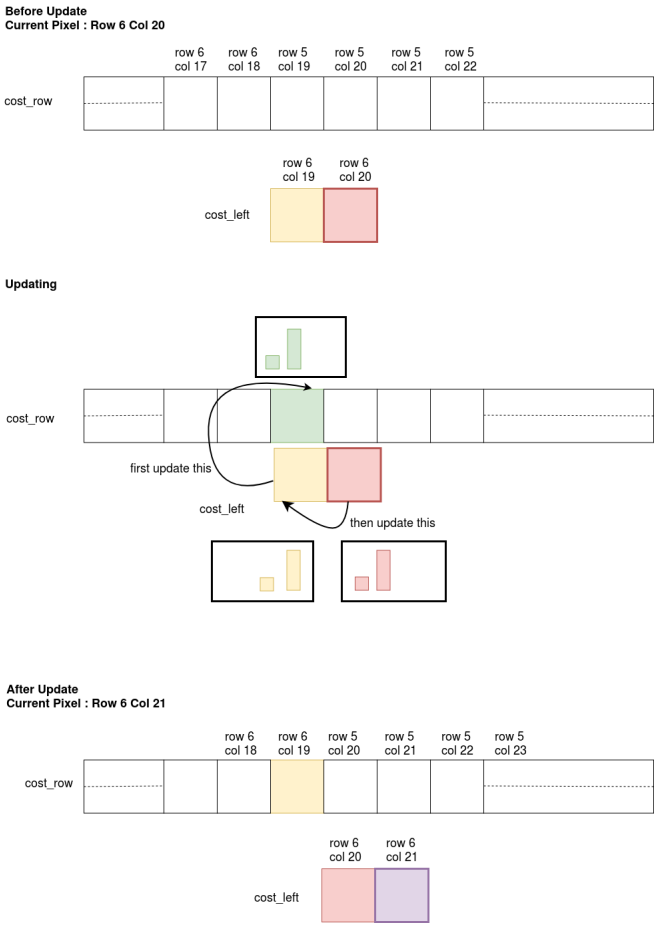


Fig. 6: SGM Array Updation.

As shown in Figure 6 the current pixel under processing is at row 6 column 20. It requires data from its 4 neighbors: row 5 column 19, row 5 column 20, row 5 column 21 and row 6 column 19.To generate data for current pixel we use the data of $cost\_left$ and 3 pixel vectors of $cost\_row$. As we compute the disparity for this pixel and also performing the housekeeping tasks of generating the required data, we update the structures as shown in Figure 6. The data from $cost\_left$ is moved to the top-left neighbour of the current pixel in $cost\_row$. The top left pixel cost data is not required anymore and hence is not stored. After this update is done, the currently generated data is moved into $cost\_left$.

Pixels at the top, left and right edge of the image are considered to have neighbors with a maximum value of aggregated cost. As SGM cost aggregation step is a minimization function, they are effectively ignored. The $cost\_row$ and $cost\_left$ structures are initialized to a maximum value before the stereo matching process. This initialization has to be done for every frame.

D. HLS Implementation

High-level Synthesis(HLS) platform such as Vivado HLS (from Xilinx) facilitates a suitably annotated description of

compute-architecture in high level language like C or C++ , which it converts to a low-level HDL based description of the same computing architecture. The generated VHDL or Verilog code is then synthesized to target fpgas. We have used Vivado HLS tools provided by Xilinx to convert our C implementation to HDL and package it to an IP for further use. The structure of HLS stereo matching code is as follows.

```
void stereo_matching_function(){
for(int row=0; row<IMG_HEIGHT; row++) {
 for(int col=0; col<IMG_WIDTH; col++) {
  //Reading pixel from DDR through AXI4
  protocol in row-major order
  //Shifting the Census Match window in
  the left and right blocks
  for(int d=0; d<SEARCH_RANGE; d++) {
   //Match l_window with r_window[d]
   //Update the min cost index
   //Add the necessary output to the cost
   row and cost left vectors
  }
 //write disparity image pixel to DDR
 }
}
}
```

There are no operations between the row and col loop, hence they can be effectively flattened into a single loop. The plan was to pipeline the merged row-column loop. Thus resulting in increase of frame rate by disparity range times if the pipeline throughput had been 1. However the resources in fpga device on Zedboard are not enough to permit the pipelining the row column loop. Hence, only the search range loop was pipelined. The arrays used in the implementation have been partitioned effectively to reduce the latency. Based on the availability of Hardware resources we have divided the whole image into sections and disparity of each section is computed in parallel. It was observed that a frame rate of 2.1 fps is obtained with the most used resource being Block RAM (BRAM) 17%. The time required for processing one frame for such an implementation can be given as

$$T \propto no.\,of\ rows\ \times no.\,of\ columns \times (search\ range\ + pipeline\ depth\ ) \tag{4}$$

The characteristic of this implementation is that the logic synthesized roughly corresponds to the matching of two Census windows, the cost aggregation arithmetic and on-chip memory to store data for the next iterations. As we sequentially iterate over rows, columns and disparity search range we reuse the same hardware. Thus, the FPGA resources required are independent of the number of rows, columns and search range but computation time required is proportional to these parameters as shown by equation 4. This gives us the idea to divide the images into a number of sections along the rows and process the sections independently by multiple such SGM blocks. As the most used resource is BRAM at 17%, we can fit 5 such SGM blocks with each block having to process 5 sections of the image i.e. 128 rows in parallel. Thus we increase resource usage 5 times and reduced the time required for computation by the same resulting in 10.5 fps.

One flaw to this approach is that if we divide the input image into exactly 5 parts, there will be a strip of width window size at the center of the disparity image where the pixels will be invalid. The solution to this is that the height of each section is $image\_height/5 + window\_size/2$. This is shown in Figure 7 for an example of 2 sections.

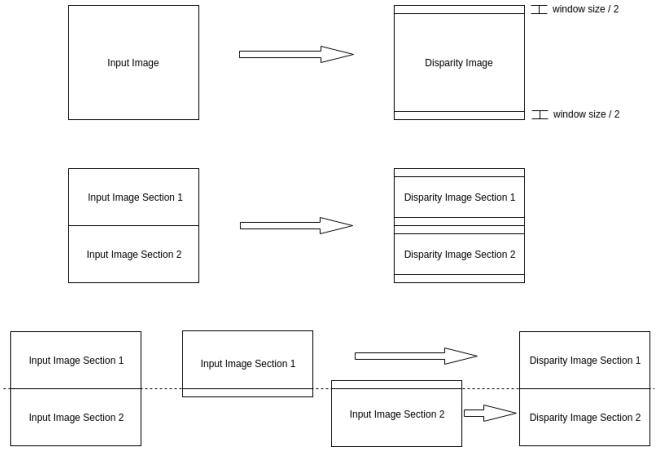


Fig. 7: Dividing the input image into two sections to be processed by two blocks simultaneously

E. Hardware Setup

Figure 8 shows the hardware setup. The Zed camera is connected to a USB 2.0 port of the Zedboard. The Zedboard is booted with petalinux through SD card. In the case where Intel RealSense camera is used, we require ethernet to receive the images. The only other connections to Zedboard are the connection to VGA display and power.

Fig. 8: Hardware setup

IV. Experimental Results and Evaluation

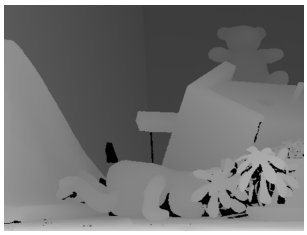
The obtained frame rate for the implemented system is 10.5 fps with Zedboard running at 100 MHz. The Power

consumption of the computation which is performed in FPGA is 0.72W whereas the on-chip arm processor which is being used to capture the images and start the FPGA peripherals along with the ELP stereo-camera consumes 1.68 watt , thereby raising consumption to 2.4W. A $10m\Omega$, 1W current sense resistor is in series with the 12V input power supply on the Zedboard. Header J21 straddles this resistor to measure the voltage across this resistor for calculating Zedboard power [10]. The resource usage is summarized in Table I. It is observed that the BRAM utilization is the most. This is due to storing large cost arrays.

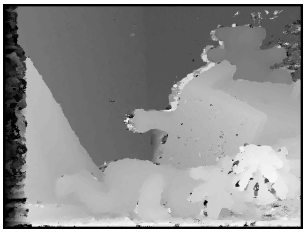
	BRAM	DSP	FF	LUT	LUTRAM
Utilization	132	65	39159	37070	981
Available	140	220	106400	53200	17400
% Utilization	94.3	29.5	36.8	69.6	5.64

TABLE I: Resource utilization for the entire design in Zedboard

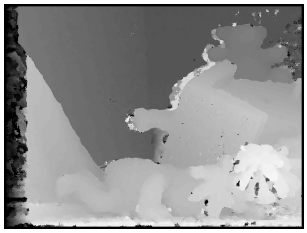
(a) Left image



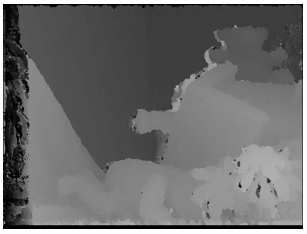
(b) Ground truth



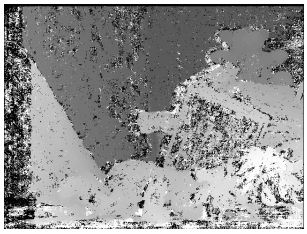
(c) SGM 4 paths software



(d) SGM 8 paths software



(e) SGM 4 paths hardware



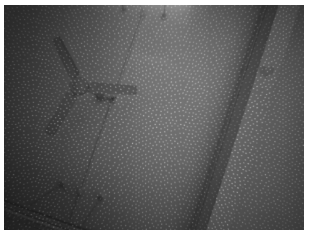
(f) SGM with arrays initialized to zeros

Fig. 9: SGM results on Middlebury images

The algorithmic performance can be measured by percentages of erroneous disparities with respect to ground truths on the Middlebury test images. A 5 pixel tolerance is considered due to intensity variation caused by changing resolution of raw image. The percentage of erroneous disparities for different images is summarized in Table II. It is notable that no post processing has been done on the SGM output.

Image	SGM
Teddy	11
Dolls	17
Books	20
Moebius	20
Laundry	27
Reindeer	27
Art	30

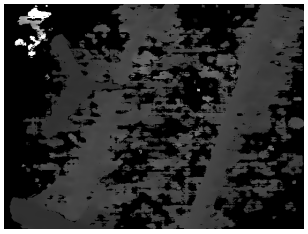
TABLE II: Percentage error of disparity image pixels as compared to ground truth for Middlebury images



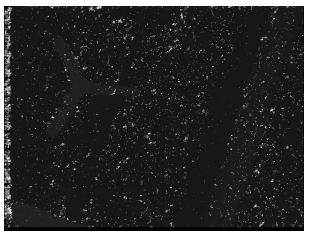
(a) Left image with IR blaster on

(b) Left image with IR blaster covered

(c) Disparity image from camera with blaster on



(d) Disparity image from camera with IR blaster covered



(e) SGM disparity image with blaster on

(f) SGM hardware disparity image with IR blaster covered

Fig. 10: SGM results on Realsense image: effect of texture

Figure 9 shows the software and hardware implementation results on Teddy image from Middlebury 2003 dataset [5]. Figure 9c-d show the results of an inhouse software implementation of SGM and Figure 9e shows the result of the hardware implementation. It can be observed 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 9f 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. Similar results

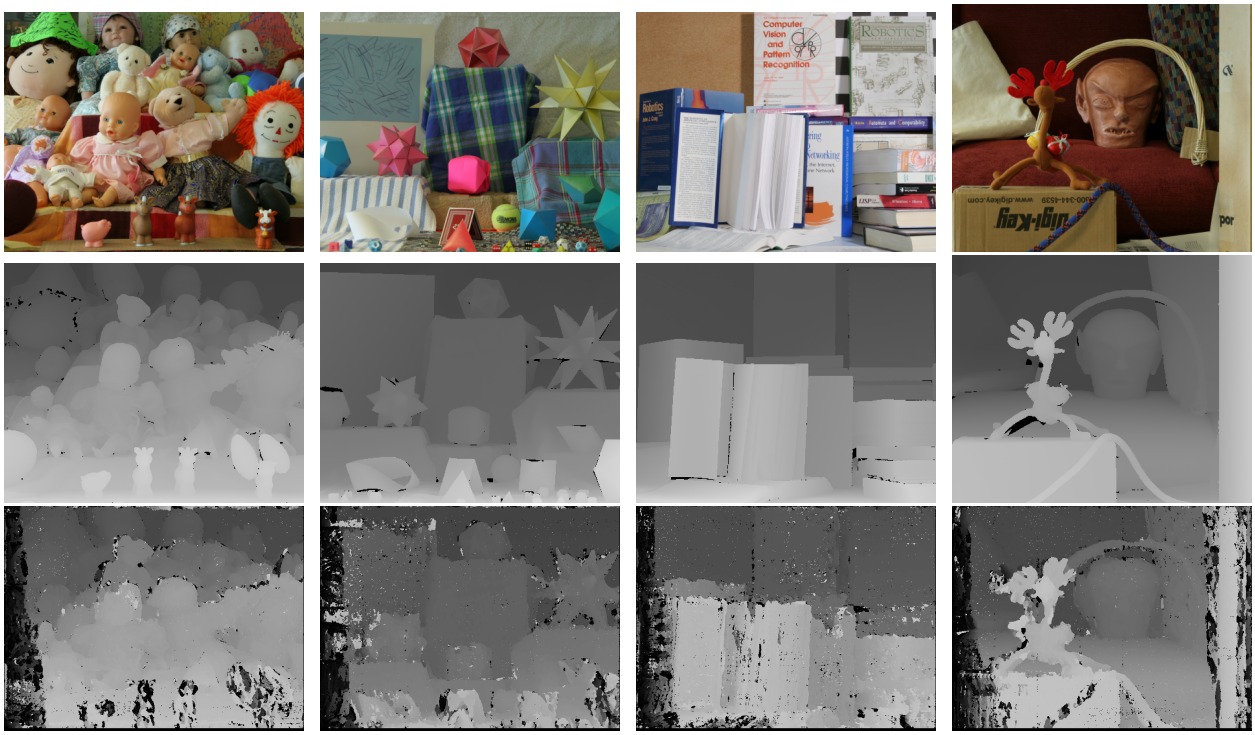


Fig. 11: Qualitative Comparison of our results with some of the Middlebury data set. 1st Row contains the Left Raw Images, 2nd Row contains the ground truth of the corresponding Images and 3rd Row contains the Output of our Implementation.

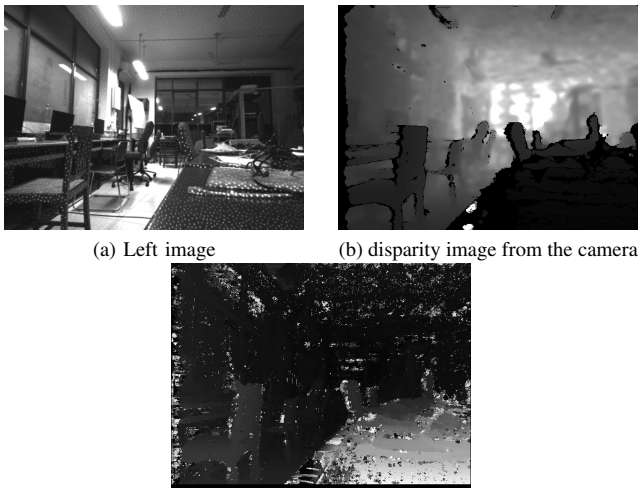
with frame rate around 8.3 fps were also achieved by an inhouse GPU implementation of SGM on Jetson TK1 board which is of MAXWELL architecture with 256 cores and power consumption < 10 watts. This implementation is analyzed and optimized by using OpenMP for multi-threading and AVX (Advanced Vector Extension) registers for vectorization. GPU shared memory is used to reduce the global memory access. CUDA shuffle instructions are used to speed-up the algorithm and vector processing is also applied.

Fig 12 and 13 shows the captured image and the corresponding disparity image obtained using the SGM implementation. The Intel RealSense camera also provides a disparity image. This is shown in Figure 13b. The convention followed here is opposite i.e closer objects appear darker.

(a) Left image classroom (b) SGM disparity image classroom

Fig. 12: SGM results on ZED camera image

The Intel RealSense camera has an infrared (IR) light projector which projects structured light onto the scene. This



(a) Left image (b) disparity image from the camera

(c) SGM hardware disparity image

Fig. 13: SGM results on Realsense image: lab

pattern can be seen in Figure 13a. Figure 10 shows the effect of the infrared projector on disparity estimation. Figure 10ace show the captured left image from the camera, disparity image obtained from the camera and the computed disparity image when IR blaster was on. Figure 10bdf show the same images when the IR blaster was covered. Incase of 10e although the image contains salt noise, it can be easily filtered out. The fan blades can be easily seen in the disparity image. In 10f there

are more number of white pixels which imply that the object is very near to the camera which is a false result. As can be seen, the structured light projector helps in stereo matching by adding texture to non-textured surfaces.

Figure 14 shows the scene and the corresponding disparity image obtained on the VGA monitor. The camera can be seen on the left side of the image.

Figure 11 shows the qualitative comparison or our results with Middlebury data set. We can see that the objects placed near are not accurate this is because we have used the disparity range of 92 pixels and so it is not able to find a match in the corresponding left and right images. Thus for a better accuracy, disparity range can be increased with the trade-off being update rate as the pipeline latency will increase.

Fig. 14: Scene and disparity image on VGA monitor

Finally we inform the reader about our comparison with R3SGM [6] work. Table III shows the comparison of hardware utilization between our approach and [6] which shows ours uses much lesser Hardware Resources and thus having less power consumption. Furthermore, if we were to use fpga used in [6], we would have far more liberty with resources that can be leveraged to further pipeline the design and obtain another order of speedup. However we have focused on very low power consumption as well as small form factor that is necessary for drones vision, blind aid etc. We can extrapolate the frame rate likely to be achieved by our design on ZC706 board as below. We can replicate the hardware four times (assuming other resources are under limit) to utilize all of the BRAM, and get 40fps performance. However, it would increase the power consumed by zynq chip, as well as by camera and DDR subsystems for this higher frame capture and processing rate.

V. Conclusion

In this paper we presented the hardware implementation of the MGM [7] which is a variant of SGM [3] on Zedboard [10] an FPGA-ARM based SOC inspired by R3SGM [6]. In order to reduce the memory consumption, we have grouped 4 paths-

	BRAM18K	DSP	FF	LUT	Frame Rate	Power (Approx)
Ours	132	65	39159	37070	10.5	0.72W
[6]	163	-	153000	109300	72	3W

TABLE III: Comparison of FPGA Hardware Resources(Approx) and power consumption between our approach and [6]

left, top left, top, and top right, whose pixel data are available while processing as a result of row-major order streaming process. The efficient utilization of hardware resources resulted in a low power consumption of 0.72W for data processing on FPGA that computes the Rectification and disparity Map generation and with 1.68W for data acquisition from Cameras along with starting the peripherals using the on board ARM processor achieving an update rate of 10.5Hz with a good accuracy as was shown in TableII and Figure11. This system is highly suitable to be used in micro UAVs, blind Aids or any portable types of equipment with a small form factor and high power constraints.

References

[1] R. Zabih and J. Woodfill [*Non-parametric local transforms for computing visual correspondence*] In Proc. ECCV, pages 151158, 1994

[2] T. Kanade [*Development of a video-rate stereo machine*] Proceedings of International Robotics and Systems Conference (IROS'95), Pittsburgh, Pennsylvania, Aug. 5-9, 1995, pp. 95-100.

[3] H. Hirschmuller [*Stereo Processing by Semiglobal Matching and Mutual Information*] IEEE Trans. Pattern Anal. Mach. Intell., 2008,30, (2), pp. 328341

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

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

[6] Oscar Rahnama, Tommaso Cavallari, Stuart Golodetz, Simon Walker and Philip H. S. Torr . [*R3SGM: Real-time Raster-Respecting Semi-GlobalMatching for Power-Constrained Systems*] International Conference on Field-Programmable Technology (FPT), Vietnam, 2018.

[7] G. Facciolo, C. de Franchis, and E. Meinhardt [*MGM: A Significantly More Global Matching for Stereovision.*] BMVC, 2015.

[8] Rostam Affendi Hamzah and Haidi Ibrahim [*Literature Survey on Stereo Vision Disparity Map Algorithms*]vol. 2016, Article ID 8742920, 23 pages, 2016.

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

[10] Zedboard datasheet:(2019,August 25) Retrieved from http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[11] Zynq 7000 datasheet:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[12] Vivado HLS user guide:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[13] Vivado Synthesis user guide:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado-synthesis.pdf

[14] XSCT reference guide:(2019,August 25) Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug1208-xsct-reference-guide.pdf

[15] Intel Realsense D435i Depth Camera:(2019,August 25) Retrieved from https://www.intelrealsense.com/depth-camera-d435i/

[16] Zed Camera:(2019,August 25) Retrieved from www.stereolabs.com

[17] OpenCV:(2019,August 25) Retrieved from https://opencv.org/