PyPix

3D scene reconstruction based on stereo vision

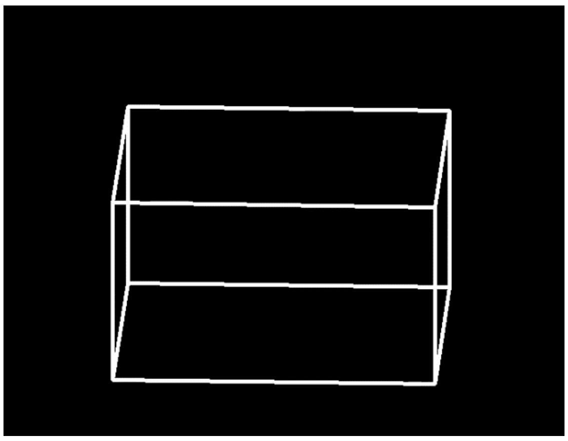
Geet Rose Jose

Spyridon Bizmpikis

James McCleary

Supervisor’s name: Louise H. Crockett

Xilinx University Program FPGA and SOC University Design contest



Department of Electronic and Electrical Engineering

University of Strathclyde, Glasgow

Table of Contents

1. Introduction

1.1. Project motivation

In this digital era, real time access to 3-dimensional visual data has immense potential. The human sensory system is wired such that interactions and remote control are easier if the digital world can be visualised as close to the real world as possible. For example, applications such as telepresence/telemedicine through AR/VR for remote surgeries, autonomous navigation of drones/ rovers, 3D gaming all require real time access to both visual and depth information. This creates an interactive experience for humans as well as bots and the applications are endless. With such huge possibilities, innovation in this domain requires rapid prototyping tools and platforms that can process complex high-density data in real time.

In this project, we aim to take a step towards real time generation of 3D vision data. FPGAs provide this design flexibility to rapidly accelerate the bottlenecks in 3D synthesis making them an obvious choice. Applications that integrate drones or head mounted devices for AR/VR, need portable systems with wireless operations hence optimised power consumption is a prerequisite. Also, it is very evident that rapid prototyping technologies and tools (Arduino, 3d printer raspberry pi) etc has been the catalyst for innovations in such domains. Hence, working with tools that provide this flexibility was a major goal. PYNQ platform supports developing applications in the Python programming language using Jupyter Notebook and hence, enabling high productivity. It has been proved to be the best choice for such applications.

In the following sections, the whole design pipeline for 3D construction and visualisation has been implemented. As the acceleration of the complete pipeline was not in the scope of this project, the individual blocks were implemented and tested in different systems. The key bottlenecks for real time 3D point generation were identified and discussed in the upcoming sections with the aim to make future complete implementation easier.

1.2. Equipment and services used

The equipment and services used for this project is listed below:

- Laptop with integrated camera for input images/video (resolution: 1280x720)
- Jupyter Notebook
- PYNQ-Z2 SoC board and peripherals

1.3. Report structure

The following sections of the report are split into 1. Background theory and 2. Methodology and experimental results associated with the background theory. A conclusion is provided at the end and the future work of this project is also discussed.

2. Background theory

In this section, the background theory that is directly related to the technical aspects of this project is discussed in detail.

2.1. Camera calibration

Any 3D reconstruction algorithm would be based to some input data, either in the form of images or video, and a suitable device for such input acquisition is a camera. Assuming a pinhole camera model, a scene is viewed by using a perspective transformation to project 3D points into the 2D plane, also known as image plane. The equation that describes the perspective transformation is the following

$$s\begin{bmatrix}u\\v\\1\end{bmatrix}=\begin{bmatrix}f_x & 0 & c_x\\0 & f_y & c_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}r_{11} & r_{12} & r_{13} & t_1\\r_{21} & r_{22} & r_{23} & t_2\\r_{31} & r_{32} & r_{33} & t_3\end{bmatrix}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}\text{, or equivalently}$$

$$sm' = A[R|t]M'$$

In the above matrix, m' represents the 2D points in the image plane (in pixels) multiplied by a scaling factor s, A is the camera matrix, $[R|t]$ is the rotation and translation matrix, and M' is the 3D object points (in mm) [1].

The process to find unknown parameters in the perspective transformation equation, assuming some parameters are known, is called camera calibration. This process can be performed only once per camera to find the intrinsic parameters or camera matrix, which can then be reused as many times as required. This occurs due to the fact that there is a single, fixed camera matrix for every single camera. In particular, the intrinsic parameters consist of the focal lengths in the horizontal and vertical directions, (f_x, f_y), and the optical centres in the horizontal and vertical directions, (c_x, c_y), and they are usually expressed in the camera matrix form $\begin{bmatrix}f_x & 0 & c_x\\0 & f_y & c_y\\0 & 0 & 1\end{bmatrix}$ [1].

The rotation and translation matrix $[R|t]$, also known as the matrix of extrinsic parameters, expresses the motion of the camera with respect to a static scene, or inversely, the motion of a scene with respect to a fixed camera. According to [1], assuming that the depth coordinate, z, in the $[R|t]$ matrix is unequal to 0, then the rotation-translation transformation can be described as

$$\begin{bmatrix}x\\y\\z\end{bmatrix}=R\begin{bmatrix}X\\Y\\Z\end{bmatrix}+t$$

$$x' = \frac{x}{z}, y' = \frac{y}{z}$$

Then, the image plane points become

$$u = f_x x' + c_x$$

$$v = f_y y' + c_y$$

Figure 1 illustrates the perspective transformation of the pinhole camera model as a 3D point is projected into the image plane.

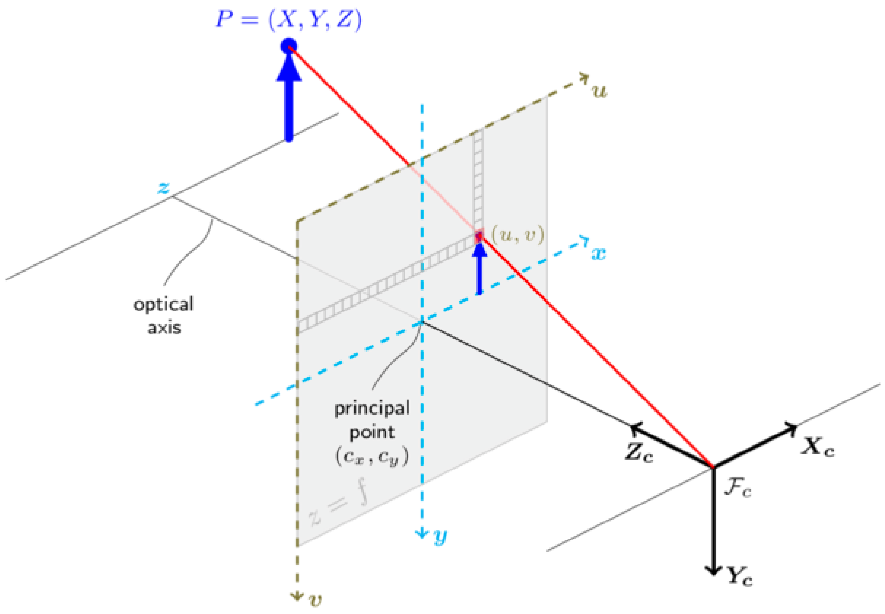


Figure 1. Perspective transformation of the pinhole camera model. Source: [1].

Once the camera matrix has been found and because it is a fixed matrix, it can be used to calculate the $[R|t]$ matrix for every scene in an input frame, or even a sequence of frames i.e. video input stream. The information of the camera matrix is core to many image-processing applications, including 3D reconstruction, pose estimation and distortion removal and therefore, its importance is highly significant.

2.2. Disparity map

Humans perceive depth information by comparing the relative shift between the two scenes seen through the two eyes. Disparity map is the technique used by vision engineers to store this depth information. It refers to the pixel difference or motion between a pair of stereo images and in the disparity-map is the representation of these measurements as intensity. This is very similar to how the eyes see depth and can be tested by closing one eye and looking through the other rapidly.

The main challenge in disparity map processing is that the number of calculations required is proportional to the number of pixels. For realistic rendering of 3D objects the Stereo cameras need to generate very high-density images. This makes the challenge to be computationally complex. Thus, for real time generation of 3D vision, the underlying hardware must be capable of handling this huge amount of data and process them in real time.

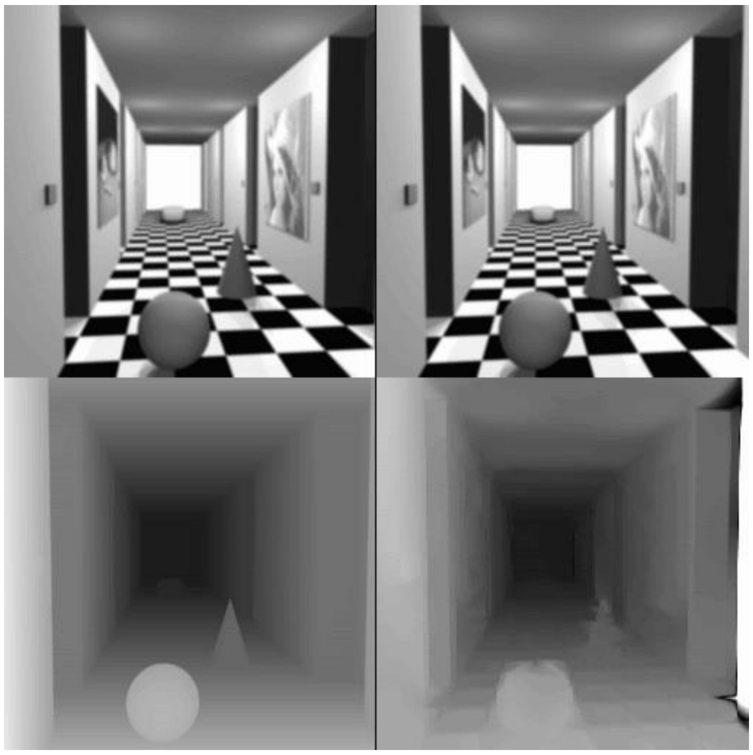


Figure 2. The objects in the foreground are brighter, denoting greater motion and lesser distance. Source: [4].

Figure 3 shows how these points are calculated mathematically. Here O & O' are the two camera centres and B is the distance between the two centres. The distance between points in the given image plane corresponding to the 3D point and camera centre is represented by x & x'. From the equation above, the depth of a point in a scene is inversely proportional to the difference in distance of corresponding image points and their camera centres. This can be used to derive the depth of all pixels in an image using direct geometric relations.

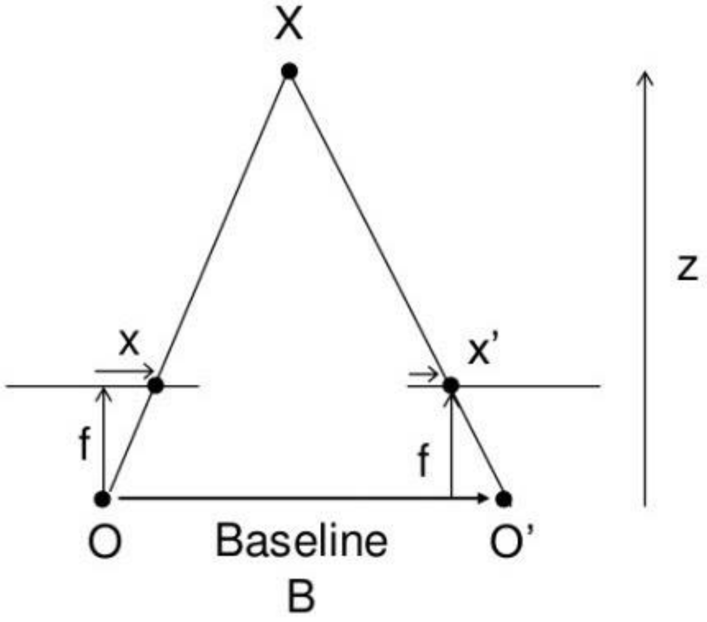


*Figure 3. Disparity=x−x' = B*f/Z. Source: [5].*

2.3. 3D point cloud

The 3D point cloud is constructed from the disparity map for visualisation purposes. A lot of algorithms are out there for this and accurate rendering is a compute-intensive task. Some popular methods with the full pipeline are shown in Figure 4.

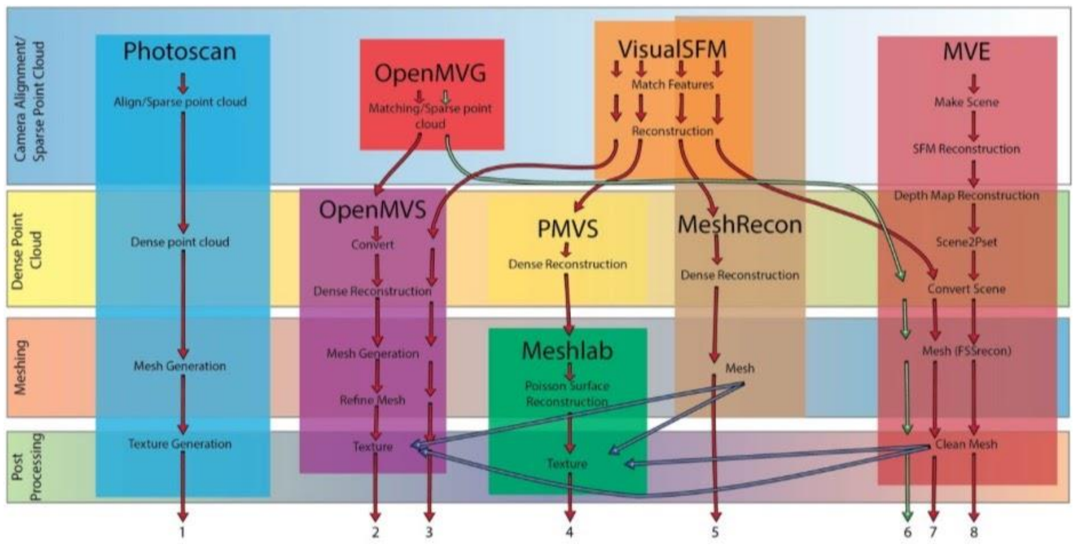


Figure 4. The full process flow from images to 3D visualisation. All the processes are done offline and the best one suggested by the author took 65 minutes. Source: [2]

It takes a significant amount of time to complete the process for high quality rendering. Even with fewer images, FPFGA's acceleration can fasten some of the blocks and provide reasonable quality. In this project, Mesh Lab, an open source software for importing and processing 3D point cloud meshes, was used for initial visualisation purposes as they are available on all platforms. As not all features of Mesh lab are required for real time visualisation. In the section below, a custom visualisation algorithm was developed aimed for real time stereo video.

2.4. 3D to 2D projection

The process of 3D to 2D projection is something that is done often on a computer. Whenever a 3D model is interacted with, the only way for a viewer to see this is for it to be projected to a 2D plane. This is basically how a camera takes a picture of the world, however in a computer with the 3D coordinates known there is a little more work to be done. To create an accurate projection, a translation of the 3-dimensional points is required to approximate their 2-dimensional location for a given view point. When this approximation has been done, an image is generated which can be viewed by the user. This still keeps the integrity of the 3D model with the image updating whenever the viewing location or angle is changed.

The principle of this projection relies on a 3x3 matrix multiplication to be performed, with an estimation of the depth being viewed and the field of view available. These parameters, along with the size of the output image, inform the program on how to approximate the 2D location of the points. The first step is to define the translation matrix. This 3x3 matrix holds the multipliers required to take the 3D coordinates and approximate the 2D ones and can be seen below.

$$\begin{bmatrix} (a)(f) & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & q \end{bmatrix}$$

Where a represents the aspect ratio, or width/height of image, f represents a factorised version of the field of view, 1/tan (FOV/2) and q represents the scaling of the z dimensions nearest and furthest points to be considered. The output of this would obviously give a 3D vector with an input of 3 points (X, Y, Z) however due to the calculation done only the first 2 points of the output are required to view the output. The third point is required to keep a track of the location of the z coordinate of that point, however since it cannot be shown on a screen it is not required for displaying the projection and can be ignored at this point.

The third dimension is required for the rotation of the object though. In this, the object can be rotated in the X, Y or Z axis, with combinations of these being achieved by performing two rotations in series. This rotation is done using a modification of the 2D rotation matrix:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

This code allows for any 2-dimensional image to be rotated on the screen. The 3-dimensional version of this is actually simple to construct, as the rotation is being performed in only 2 of the 3 dimensions, the same amount as above. This means that for the axis this rotation is being performed around, no change will be required. This gives 3 separate rotation matrices:

X - 3D rotation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Y - 3D rotation matrix

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Z - 3D rotation matrix

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To have a completely static image, a constant value would be used for theta, however by changing the value of theta over time the object can be rotated or viewed from different angles easily.

To get the result, the (X, Y, Z) coordinates are multiplied by the desired matrix. All rotation calculations need to be done before the projection as doing the reverse will have the wrong 2D projection coordinates for the desired rotation.

2.5. Overlay design

Hardware acceleration is a technique that is used to increase the throughput and decrease the latency of a system by offloading compute-intensive tasks to hardware, as opposed to execute these tasks using software on a processor such as for instance, a Central Processing Unit (CPU).

The benefits of hardware acceleration include but are not limited to faster processing based on parallelism, low-power consumption, efficient utilisation of resources within the IC area and increased bandwidth. Using hardware-acceleration is particularly useful for real-time applications where high performance at a low cost is required. Such applications consist of frequent non-sequential repetitive tasks which can be run concurrently hence, utilising the parallelism of the accelerator. On the other hand, once the design has been implemented in hardware, it is not as easy to update the functionality of the design as it would be by simply using software.

FPGAs are suitable devices for accelerating functions in hardware because of the parallel architecture they consist of. In the context of PYNQ, this is achieved by designing overlays. Figure 5 illustrates the parallelism of the FPGA.

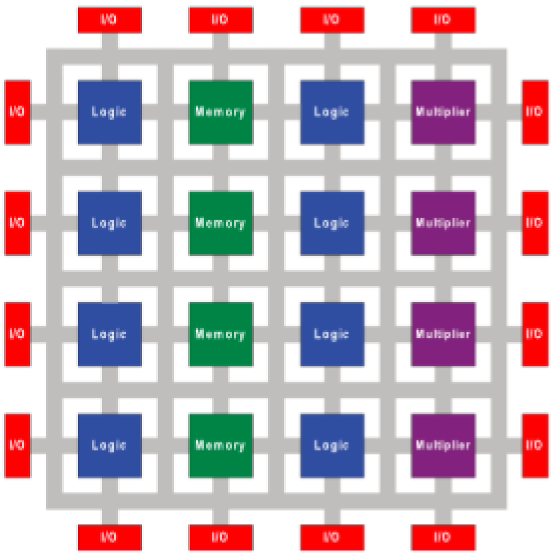


Figure 5. FPGA parallel architecture. Source: [3].

3. Methodology and experimental results

In this section, the methodology followed to implement the various stages of the proposed system and the experimental results are discussed in detail.

3.1. Camera calibration

To calibrate the camera in order to find the intrinsic parameters, OpenCV code written in Python has been used, which is provided in Appendix and has been slightly modified with respect to the original code provided by OpenCV.

According to OpenCV, the algorithm currently works based on the following 3 possible patterns

- Black and white chessboard pattern
- Symmetrical chessboard pattern
- Asymmetrical chessboard pattern

In this project, the black and white chessboard pattern has been used. It consists of 9 inner corners in the horizontal direction and 6 inner corners in the vertical direction, and it is illustrated in Figure 6.

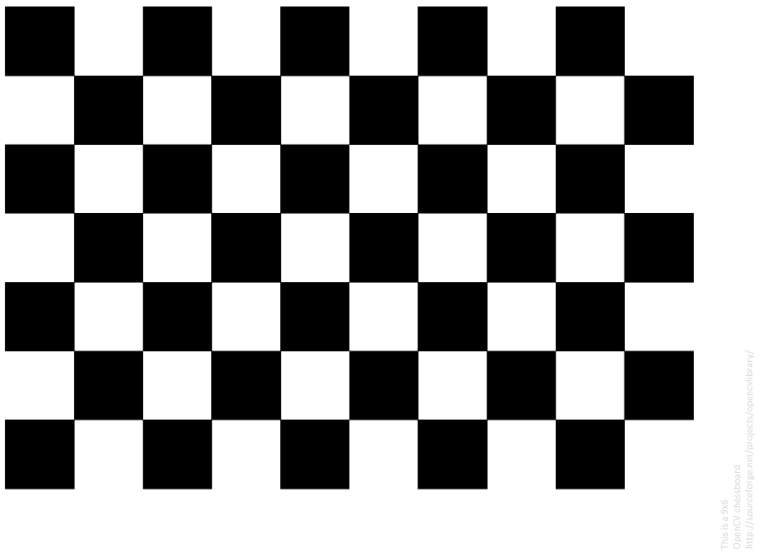


Figure 6. Chessboard pattern image for camera calibration. Source: [1].

In fact, 14 images taken from different perspectives/angles have been used as input to the camera calibration algorithm. Each of these images produces a single equation and hence, 14 images result in a system of 14 equations. It should be noted that the larger the number of pattern input images, the better the calibration result.

During runtime, the algorithm detects the inner corners of the pattern and calculates the intrinsic parameters and the distortion coefficients. It also calculates the rotation and translation vectors with respect to the camera, separately as opposed to a joint rotation-translation matrix, for each single input image. This is illustrated in Figure 7.

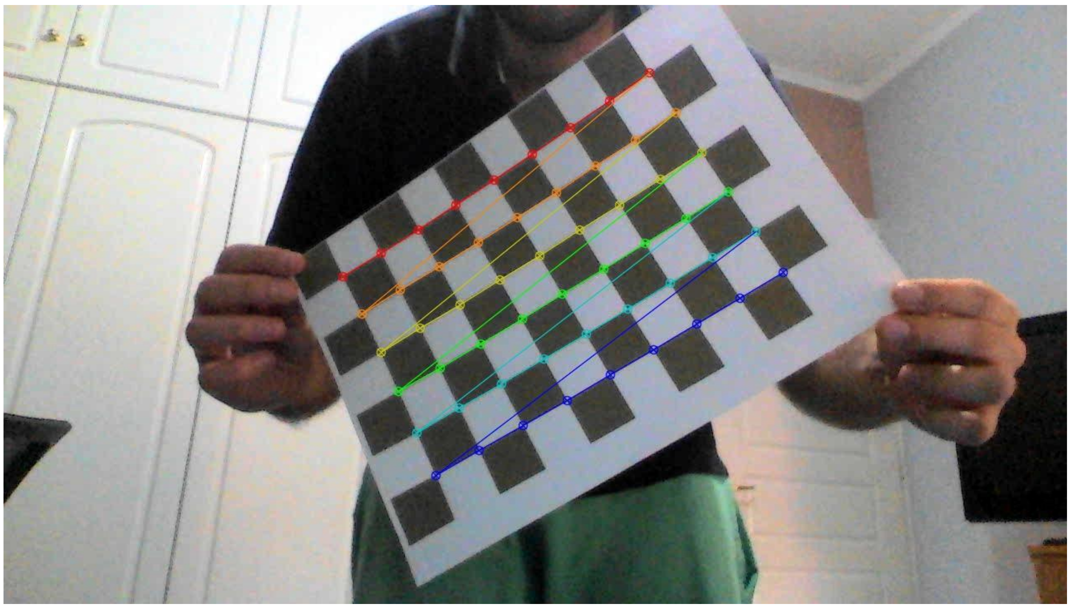


Figure 7. Pattern found during runtime.

The output of the camera matrix can be shown in Figure 8. As can be seen in this figure, the focal lengths pair (f_x, f_y) equals $(1057.4,1070.4)$ and the principal points pair equals $(636.9,250.8)$.

```
[[1.05740330e+03 0.00000000e+00 6.36995690e+02]
 [0.00000000e+00 1.07042864e+03 2.50839112e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

Figure 8. Camera matrix.

3.2. Disparity map

After camera calibration, disparity map is the next step in the design flow. The disparity map is created and timed both in the PYNQ board and a personal computer for validation. A quick summary of the whole process is shown in Figure 9.

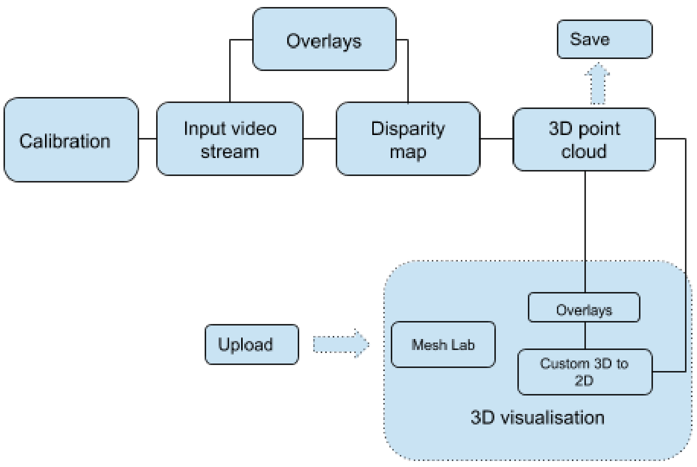


Figure 9. Design flow diagram.

The initial aim was to generate stereo vision from two cameras. Due to lack of direct access to stereo cameras a work around was tried to generate pseudo stereo video. The input video is split into two parts to virtually realize stereo vision and hence generate the disparity map. Though the generated map wouldn't be very accurate, the purpose here was to evaluate how 3D construction algorithms work on real time stereo input. The OpenCV inbuilt functions that generate disparity maps were implemented. They have been used widely and are very stable so the purpose was to accelerate the pipeline and not reinvent the wheel.

The disparity maps generated were not very accurate as was anticipated. Also, for the video input, a lot of maps were generated which would make the process really heavy during real world implementation.

In future work, a method to compress the input frame will be used or existing 3D cloud generating algorithms (VisualSFM, OpenMVS) etc will be used. There are a lot of such tools available but they are computationally intensive and only run post recording, not in real time. This proves the need for FPGAs to accelerate the process.

To test quality stereo images, the Aloe Vera stereo image dataset was used. The process was timed on the PYNQ board and it took 12.531396 seconds so this is not feasible for real time video. A custom overlay will be designed in the future to resolve this. But the disparity map was generated both on the board and the PC to validate the accuracy and as can been seen in Figure 10.

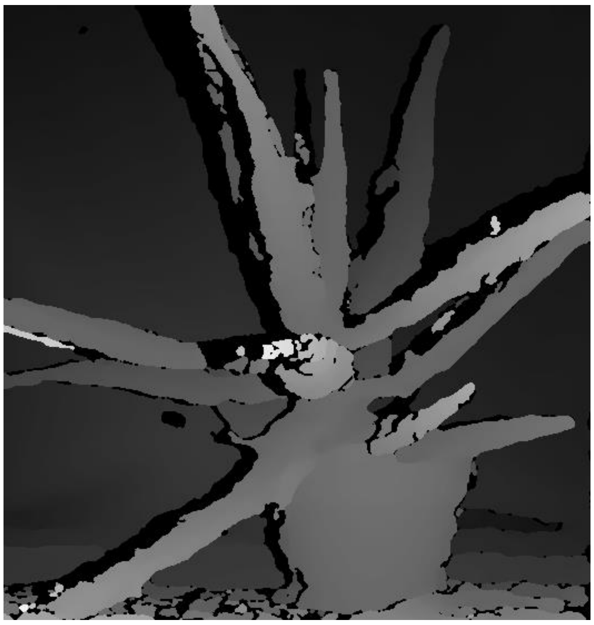


Figure 10. Disparity map generated based on stereo vision.

The next step in the pipeline was to generate the 3D point cloud and visualize the model.

3.3. 3D point cloud

The 3D point cloud was generated using the built in OpenCV function. For comparison with the custom method implemented in the next section, Mesh Lab was used to visualize the 3D point cloud. As can be seen from the figure below, the rendering was accurate. The 3D point cloud generation using the PYNQ processor took 42.973997 seconds. The result is shown in Figure 11.

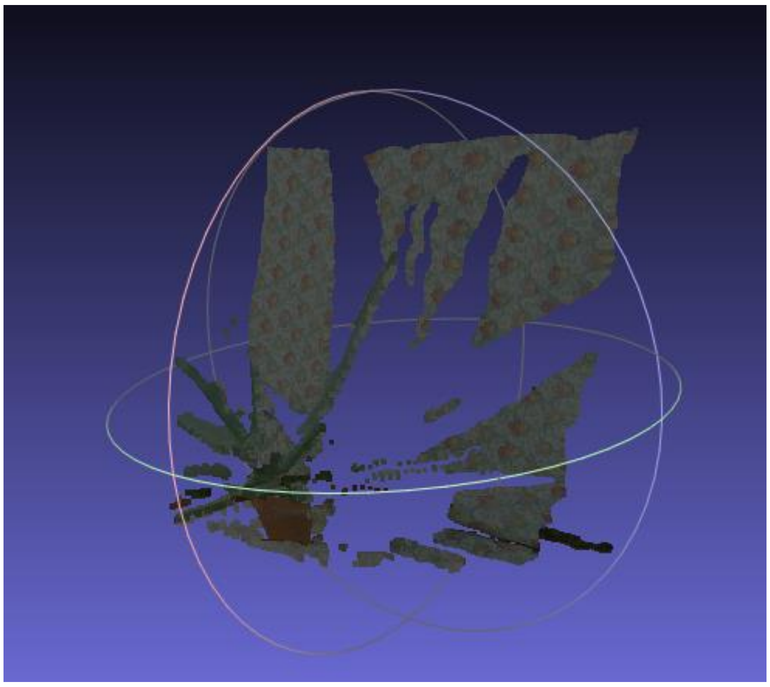


Figure 11. A view of 2D projection from 3D point cloud generated using Mesh Lab.

This proved that disparity map and point cloud generation are bottlenecks that if accelerated would make real time 3D modelling possible. Creating an overlay for these two OpenCV functions would be also part of the future work.

3.4. 3D to 2D projection

The method of creating the projected 2D image first starts with the generated 3D dense point cloud. This 3D point cloud can be treated as all of the required information about the model, as can be seen in Figure 12. They hold the positional information; X, Y, Z coordinates, as well as the colour information; Red, Green, Blue intensity. To properly test the function, a simple cube was generated with 8 points representing each of the corners on the object. This structure was then used for the rotation and translation functions before the projection being shown. In this instance, the points were connected by lines as it made visualising the output easier.

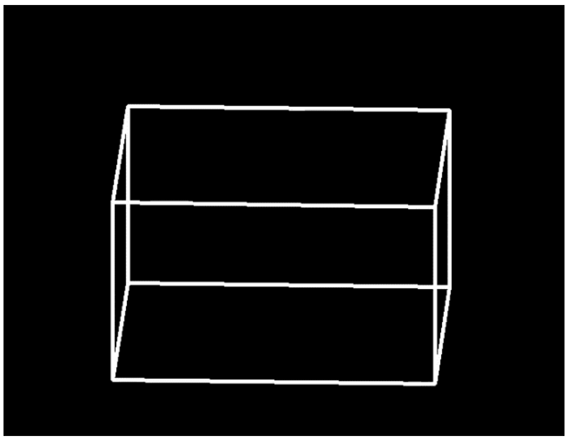


Figure 12. 3D cube being rotated and projected onto 2D plane.

This output is shown using the OpenCV library for python. This gives the user the ability to print lines or points to the screen. With the cube rotating in all 3 axes independently and the ability to output all points to the screen in the desired position and colour. The previously generated 3D dense point cloud provides the X, Y, Z coordinates and the RGB colours for each point so the rotation and translation matrices can be directly applied to this. For the current implementation, it was decided that a single input frame would be used and would be rotated to view the projection in a series of output frames. This could be changed to be a stream of input images with the view either constant or even rotating around it. With the transforms completed, the final output frames are saved to a variable. It is entirely possible to directly view the output frames as they are being generated, though on a laptop this results in approximately a new frame every 3-5 seconds. The PYNQ board was unable to interface with the OpenCV imshow functionality and therefore could not directly display the output. To keep the operation similar across all implementations, the output would be saved to a variable each time with the time taken to generate this output being tracked.

The example image holds over 230,000 individual points to be calculated. Using a laptop with an Intel i5 processor, a rotation in 1 axis followed by a projection takes approximately 102.5 seconds to produce 10 output frames. When running on the PYNQ processor, the speed was markedly slower and so the problem had to be significantly rescaled. For this, the problem size would be reduced to 2300 points for the input. This gave an acceptable timeframe for the process to run and required 14.7 seconds for a single output frame. an example of the output can be seen in Figure 13.

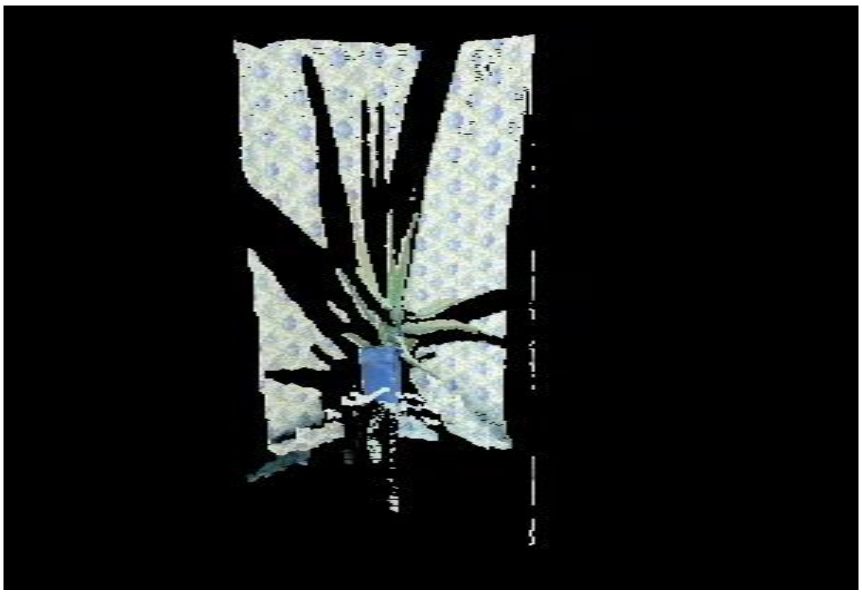


Figure 13. Example image being rotated and projected onto 2D plane.

This gives us our baseline performance to compare hardware acceleration using the DSP blocks available on the FPGA in the next section. With the working code now implemented in hardware using DSP blocks to perform the matrix multiplication in parallel it is now possible to compare the performance of the overlay. Using the same base parameters as the PYNQ processor, the overlay was able to complete an output frame in 47.5 seconds. This is a dramatic reduction in speed over using only the processor on the board, even though the multiplications in the code have been accelerated. These values also come on a data set with 100x less points than the one used for the laptop. Even with this small dataset the board was still found to be slower.

There are a couple of reasons that we suspect for this low performance. Firstly, there is no data handling procedure written for this code. This means that the data is fetched linearly for the board rather than making use of parallel buffers. This could explain why the board processor was faster than making use of the DSP blocks. The processor has the data stored right next to it and so the communication time for this reduces dramatically when compared to the DSP blocks with only a single input point. Secondly, the overlay was only computing in parallel the 3x3 matrix multiplication for a single operation. This meant that the 27 DSP blocks used for this project were all in use at the same time, but to complete a single rotation and projection they would need to be reused for each calculation individually. There was also only processing performed for each point individually, no points were calculated in parallel.

3.5. Overlay design

As mentioned previously, the main purpose of accelerating Python functions in hardware, using an overlay in the case of PYNQ, is to reduce the processing time due to the computations involved. In this project, there are 3 major compute-intensive tasks:

1. Conversion of the frames from the RGB colour space to grayscale.
2. Calculation of the disparity map based on the input frames.
3. Generation of the 3D point cloud (3D model) based on the disparity map.

As a result, an overlay has been designed, for task 1 above, to accelerate the frame conversion from RGB to grayscale. Figure 14 illustrates the processes involved until the disparity map is calculated, for 2 images as input.

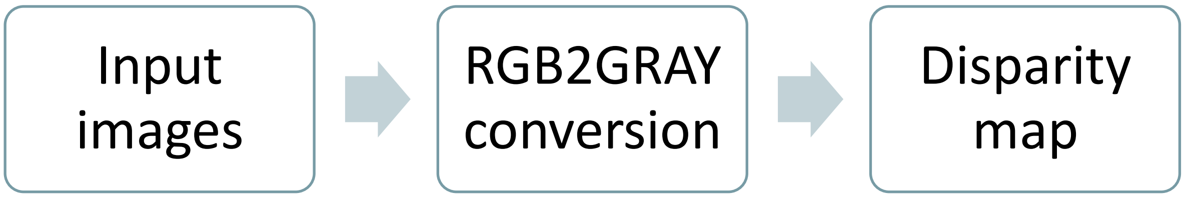


Figure 14. RGB2GRAY pipeline for input images.

Similarly, Figure 15 shows the processes from input to disparity map calculation, but this time for video input i.e. a sequence of images.

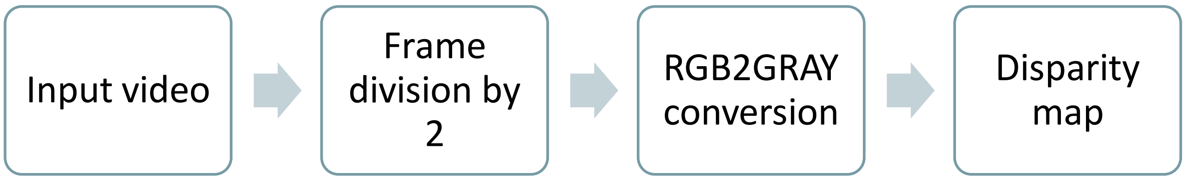
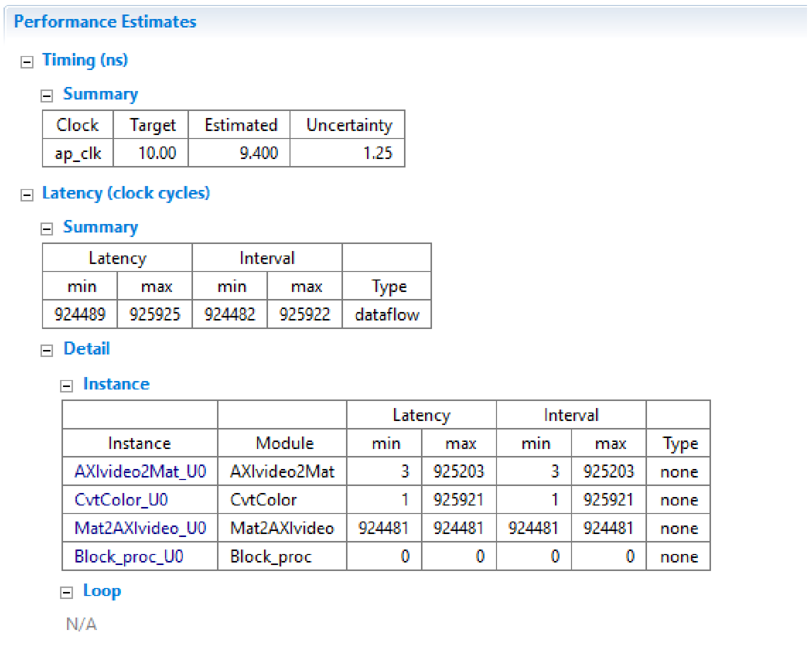


Figure 15. RGB2GRAY pipeline for input video.

The first step in designing the PYNQ overlay is to describe it using the C programming language in Vivado High-Level Synthesis (HLS). The code is provided in Appendix. In particular, the AXI interface is used for the communication between the PS and PL of the PYNQ-Z2 board and the processing is achieved by first converting the input stream from AXI video to Mat, and then after the processing, converting Mat back to AXI video for further processing.

The next step is to synthesize the code in Vivado HLS. Synthesis produces a summary for analysing performance, resource utilisation, temperature etc. The performance report is illustrated in Figure 16. In this figure, the latency is shown for the different processes involved. A report for the resource utilisation is provided in Figure 17. Clearly, for this design there has been used 842 FFs, 1558 LUTs and 3 DSP48E blocks in total. This approximates to ~0%, 2% and 1% of total resource utilisation, respectively.

Finally, the code is then exported to RTL, either in Verilog or VHDL, so the design can be imported in Vivado.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.400	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
924489	925925	924482	925922	dataflow

Detail

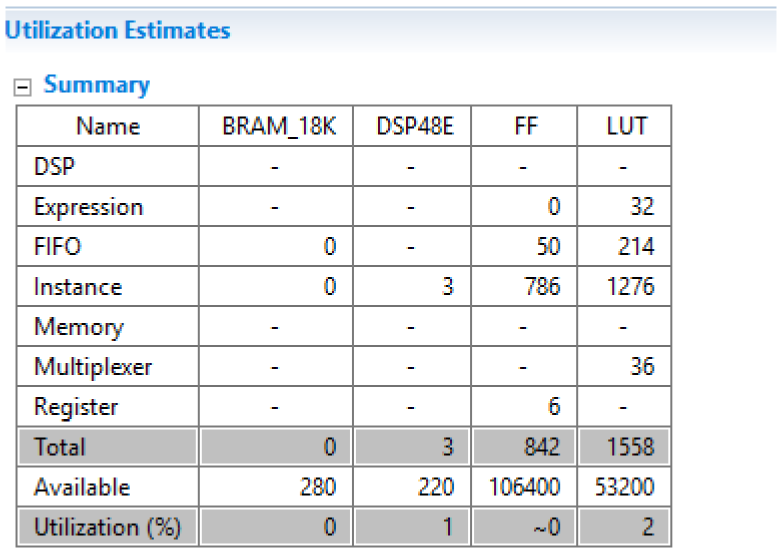
Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
AXIvideo2Mat_U0	AXIvideo2Mat	3	925203	3	925203	none
CvtColor_U0	CvtColor	1	925921	1	925921	none
Mat2AXIvideo_U0	Mat2AXIvideo	924481	924481	924481	924481	none
Block_proc_U0	Block_proc	0	0	0	0	none

Loop

N/A

Figure 16. Performance estimates for RGB2GRAY overlay.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	32
FIFO	0	-	50	214
Instance	0	3	786	1276
Memory	-	-	-	-
Multiplexer	-	-	-	36
Register	-	-	6	-
Total	0	3	842	1558
Available	280	220	106400	53200
Utilization (%)	0	1	~0	2

Figure 17. Utilization estimates for RGB2GRAY overlay.

In Vivado, the design is imported and connected to the ZYNQ7 processing system to create the interface between the PS and the PL. This is achieved using the IP Integrator tool of Vivado. The ZYNQ7 processing system already exists as an IP block, the overlay is then added as an IP to the design and connected to the ZYNQ7 PS. The schematic diagram is illustrated in Figure 18.

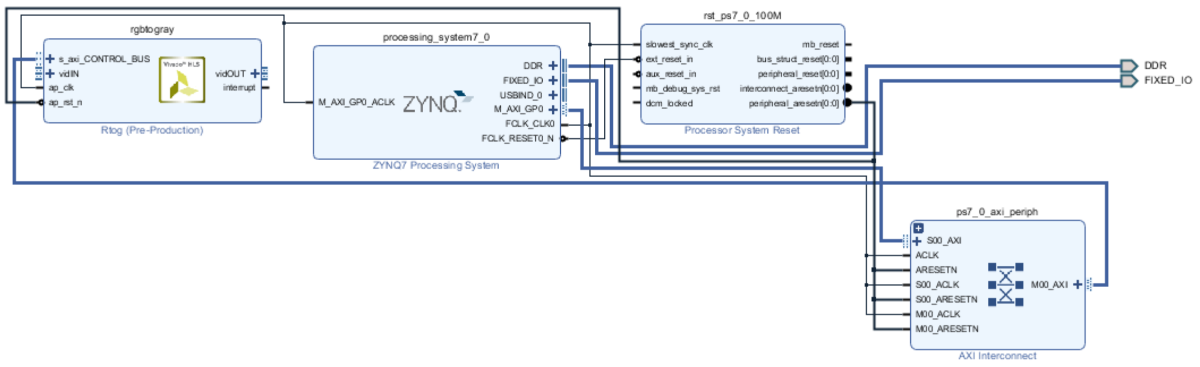
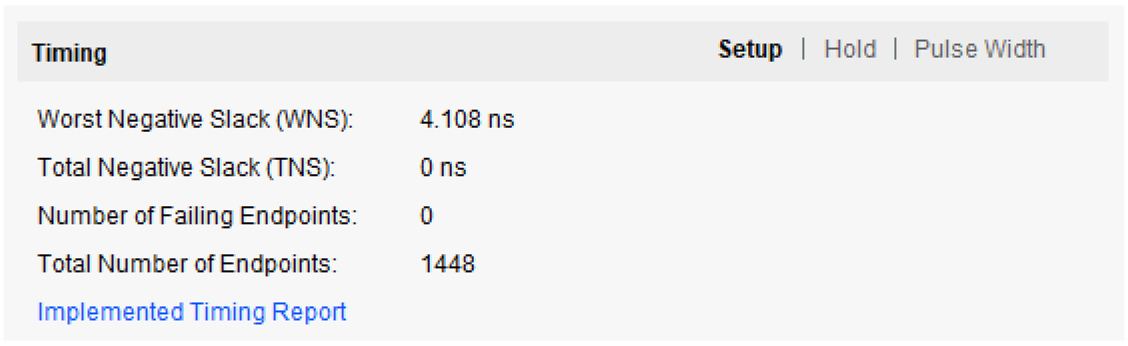


Figure 18. Block diagram including the RGB2GRAY overlay in Vivado.

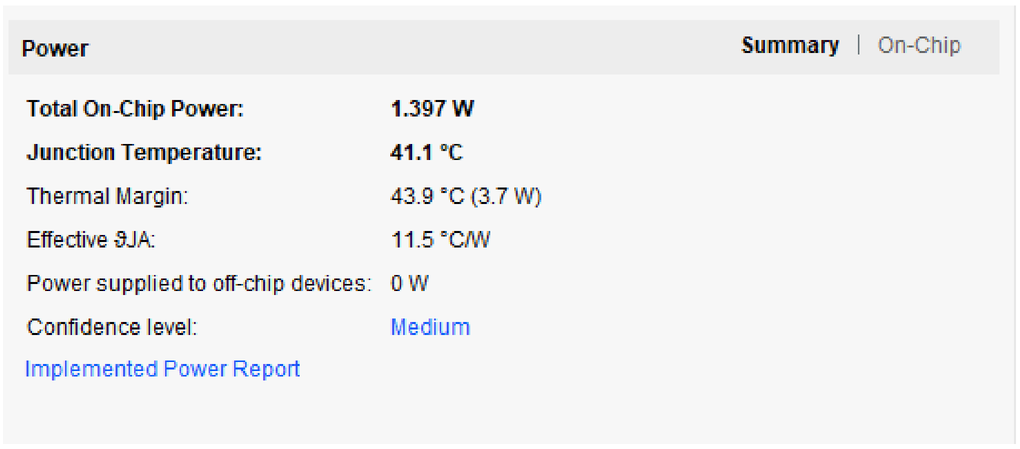
The updated design i.e. including the ZYNQ7 PS, was then synthesized in Vivado. Information such as timing and temperature for this design was produced as a result of synthesis and it is shown in Figures 19 and 20, respectively.



Timing	Setup | Hold | Pulse Width
Worst Negative Slack (WNS):	4.108 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	1448

Implemented Timing Report

Figure 19. Timing results for RGB2GRAY overlay.

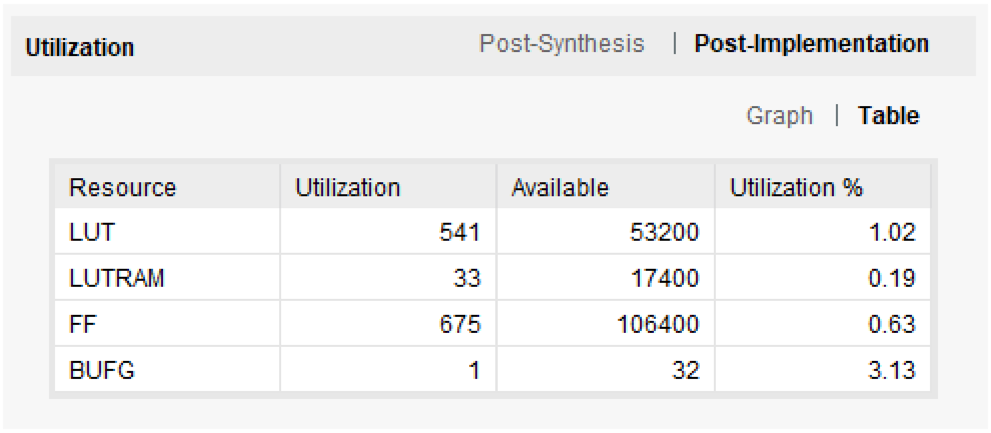


Power	Summary | On-Chip
Total On-Chip Power:	**1.397 W**
Junction Temperature:	**41.1 °C**
Thermal Margin:	43.9 °C (3.7 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Figure 20. Power consumption results for RGB2GRAY overlay.

A table describing the resource utilisation and the corresponding graph (in %), both with respect to the total device resources, are provided in Figures 21 and 22, respectively.


Utilization
Post-Synthesis
Post-Implementation
Graph
Table

Resource	Utilization	Available	Utilization %
LUT	541	53200	1.02
LUTRAM	33	17400	0.19
FF	675	106400	0.63
BUFG	1	32	3.13

Figure 21. Resource utilization for RGB2GRAY overlay (table).

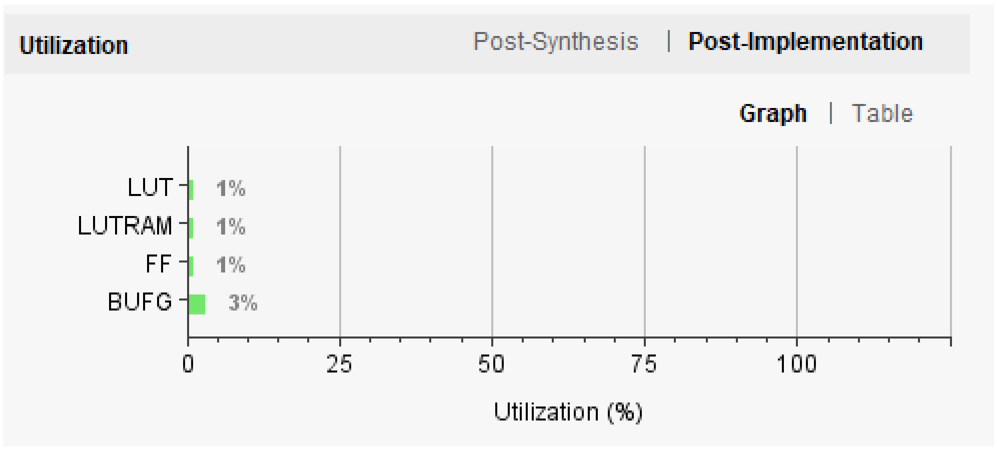


Figure 22. Resource utilization for RGB2GRAY overlay (graph).

Finally, a visualization of the resources used within the device is shown in the implemented system in Figure 23. Clearly, very few FPGA resources have been used for this design. It is possible that by increasing the number of utilized resources, further acceleration can be achieved.

It should be noted that the compute-intensive tasks 2 and 3, can also be accelerated with overlays in PYNQ, however, designing such overlays is far more complex than designing the RGB2GRAY overlay because of the many subtasks and function dependencies involved. Some of the IPs for these tasks do not already exist and custom ones should be designed. In fact, if these 2 overlays i.e. the disparity map calculation and the 3D point cloud generation, were designed and integrated to the system, real-time 3D scene reconstruction would have been achieved. Although this is possible, it is very time consuming to design and due to lack of time, the authors of this report have not worked on that.

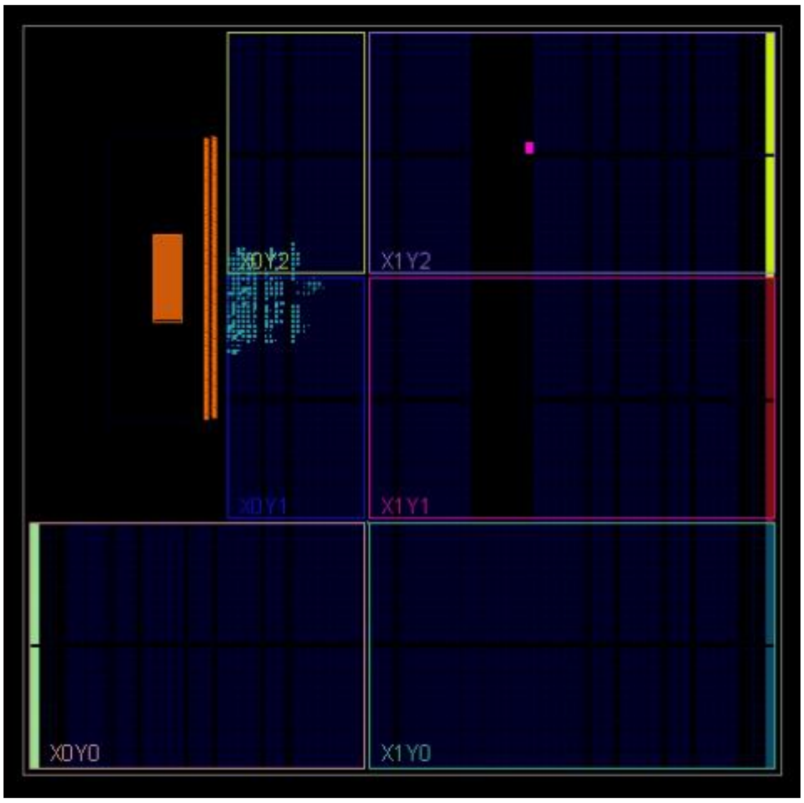
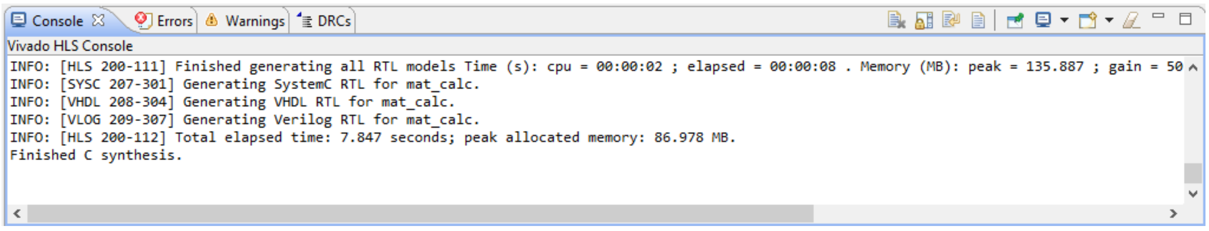


Figure 23. Visualization of utilized resources in the FPGA for RGB2GRAY overlay.

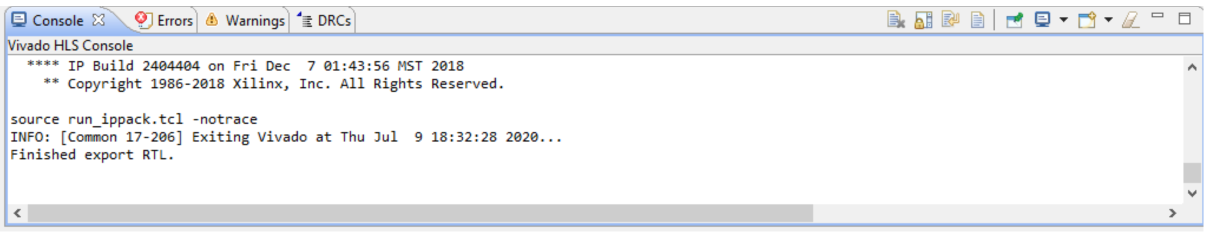
As of the MACs involved when converting the 3D model into 2D projections for visualization, another overlay has been designed. The code is provided in Appendix.

After the overlay code was written in Vivado HLS using C, the design was synthesized and the RTL code was exported, as can be seen in Figures 24 and 25, respectively.



```
INFO: [HLS 200-111] Finished generating all RTL models Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Memory (MB): peak = 135.887 ; gain = 50
INFO: [SYSC 207-301] Generating SystemC RTL for mat_calc.
INFO: [VHDL 208-304] Generating VHDL RTL for mat_calc.
INFO: [VLOG 209-307] Generating Verilog RTL for mat_calc.
INFO: [HLS 200-112] Total elapsed time: 7.847 seconds; peak allocated memory: 86.978 MB.
Finished C synthesis.
```

Figure 24. C overlay synthesis.

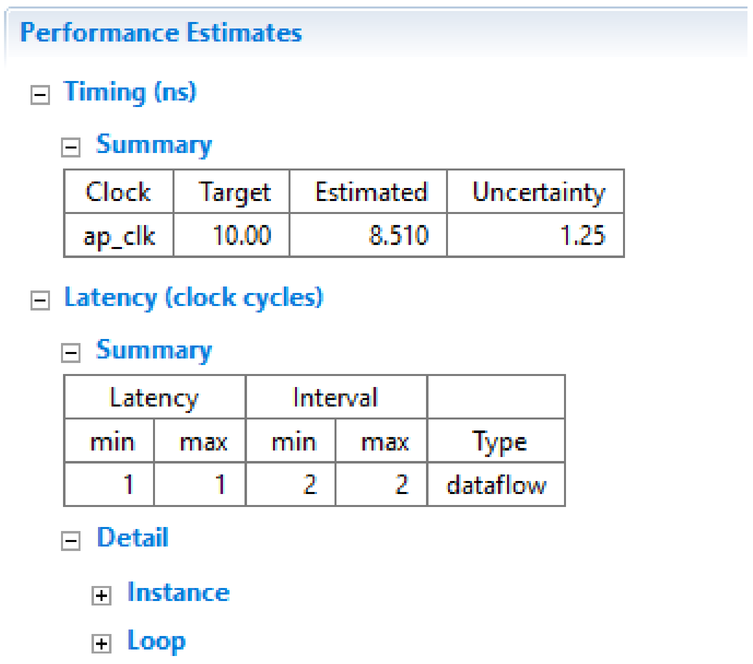


```
  **** IP Build 2404404 on Fri Dec  7 01:43:56 MST 2018
    ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
INFO: [Common 17-206] Exiting Vivado at Thu Jul  9 18:32:28 2020...
Finished export RTL.
```

Figure 25. RTL export in Verilog.

The performance and utilization estimates are illustrated in Figures 26 and 27, respectively.


Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

⊟ Latency (clock cycles)

⊟ Summary

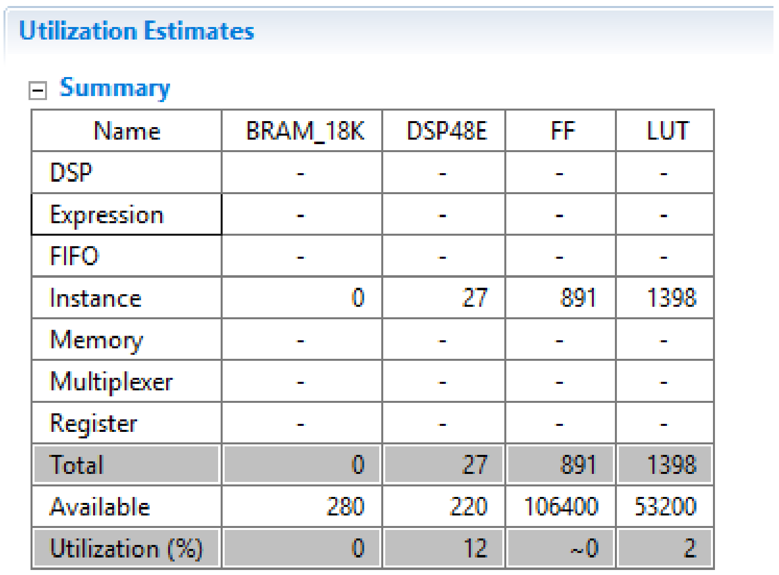
Latency		Interval		
min	max	min	max	Type
1	1	2	2	dataflow

⊟ Detail

⊞ Instance

⊞ Loop

Figure 26. Performance estimates for the mat_calc overlay.

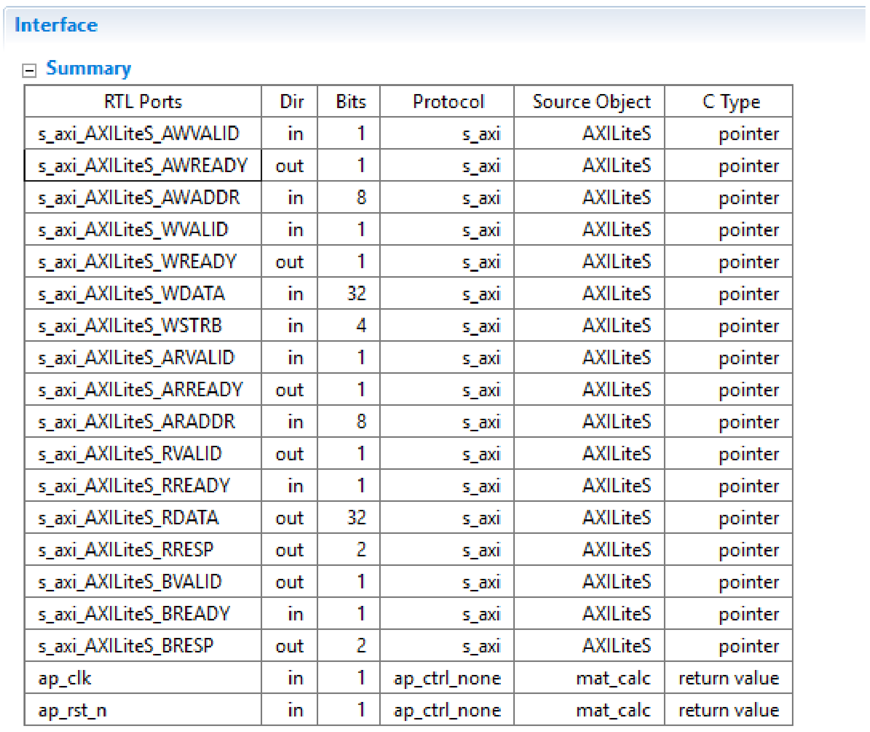

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	27	891	1398
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	27	891	1398
Available	280	220	106400	53200
Utilization (%)	0	12	~0	2

Figure 27. Utilization estimates for the mat_calc overlay.

Furthermore, the RTL ports and associated information are shown in Figure 28. For the generated data signal AXI interface ports, see Appendix.


Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_AWADDR	in	8	s_axi	AXILiteS	pointer
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	pointer
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	pointer
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_ARADDR	in	8	s_axi	AXILiteS	pointer
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	pointer
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	pointer
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	pointer
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	pointer
ap_clk	in	1	ap_ctrl_none	mat_calc	return value
ap_rst_n	in	1	ap_ctrl_none	mat_calc	return value

Figure 28. RTL ports utilized for the mat_calc overlay.

The IP design was then imported in Vivado as a block diagram, in which the. tcl and .bit files were generated. These files are required to be uploaded in Jupyter Notebook so the overlay can be properly used there.

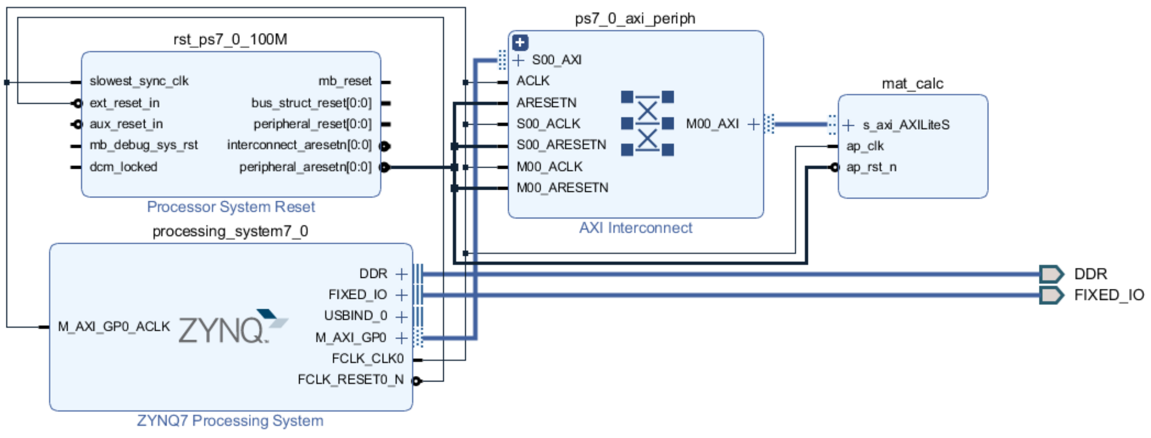
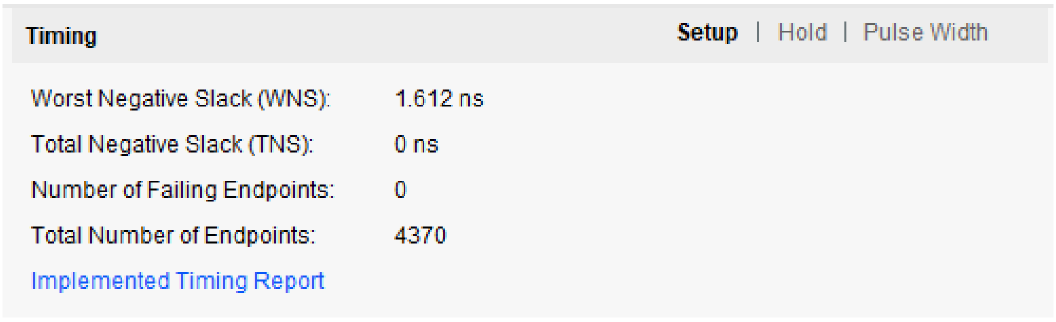


Figure 29. Block diagram including the mat_calc overlay in Vivado.

As with the RGB2GRAY overlay, the timing and power consumption information for the synthesised design are illustrated in Figures 30 and 31, respectively.

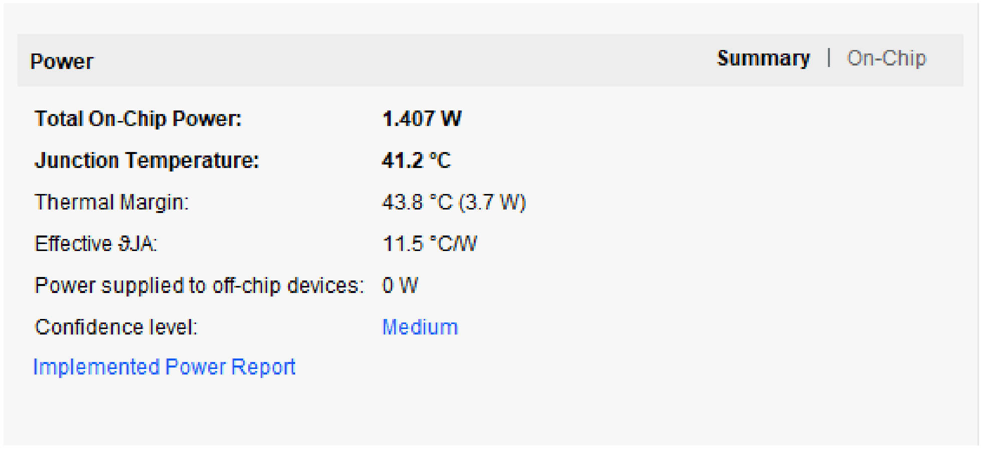


Timing | **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS):	1.612 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	4370

Implemented Timing Report

Figure 30. Timing results for mat_calc overlay.



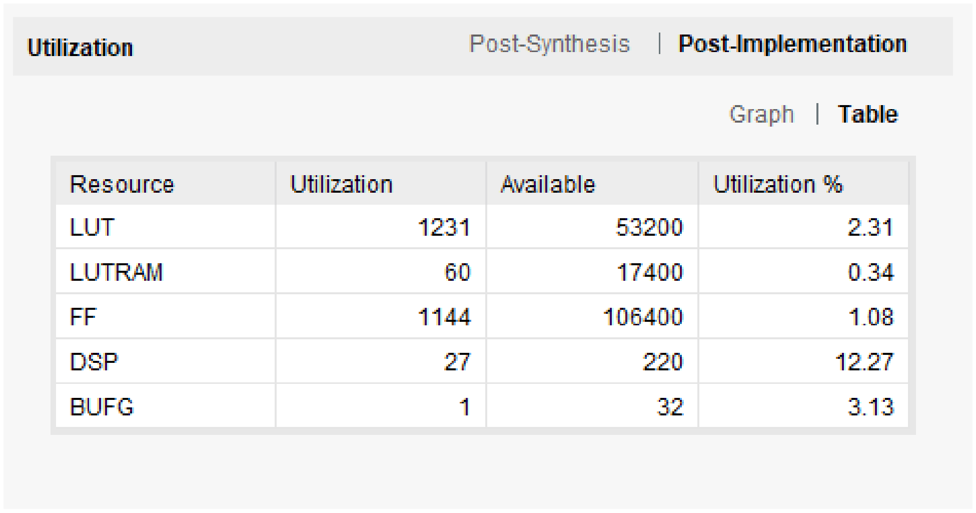
Power | **Summary** | On-Chip

Total On-Chip Power:	**1.407 W**
Junction Temperature:	**41.2 °C**
Thermal Margin:	43.8 °C (3.7 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Figure 31. Power consumption results for mat_calc overlay.

The resource utilization with respect to the total available resources is illustrated in the form of 1. Table and 2. Graph in Figures 32 and 33, respectively.



Utilization | Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1231	53200	2.31
LUTRAM	60	17400	0.34
FF	1144	106400	1.08
DSP	27	220	12.27
BUFG	1	32	3.13

Figure 32. Resource utilization for mat_calc overlay (table).

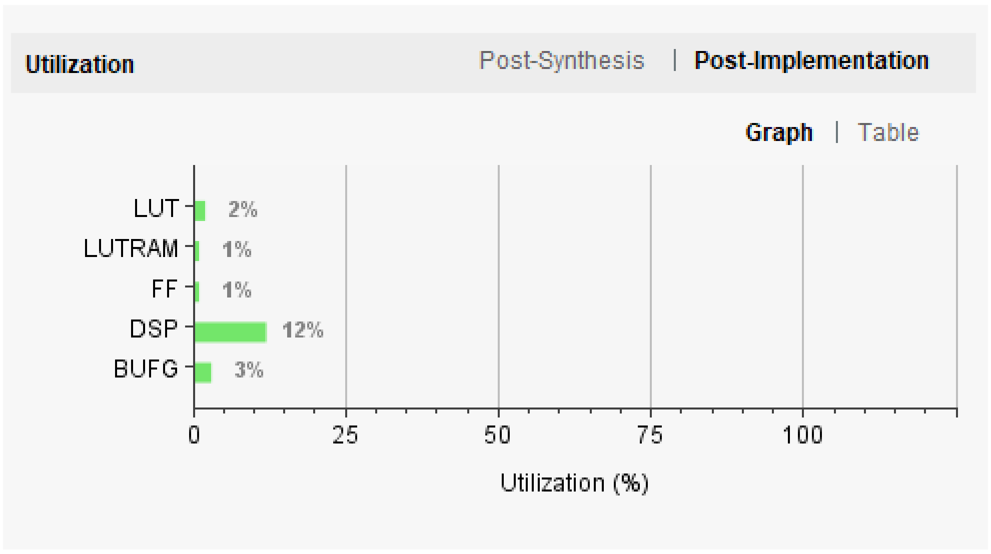


Figure 33. Resource utilization for mat_calc overlay (graph).

Finally, the data contained in Figures 32 and 33, are visualised in the implemented design on the FPGA in Figure 34. The SoC part is presented in the upper left corner of this Figure.

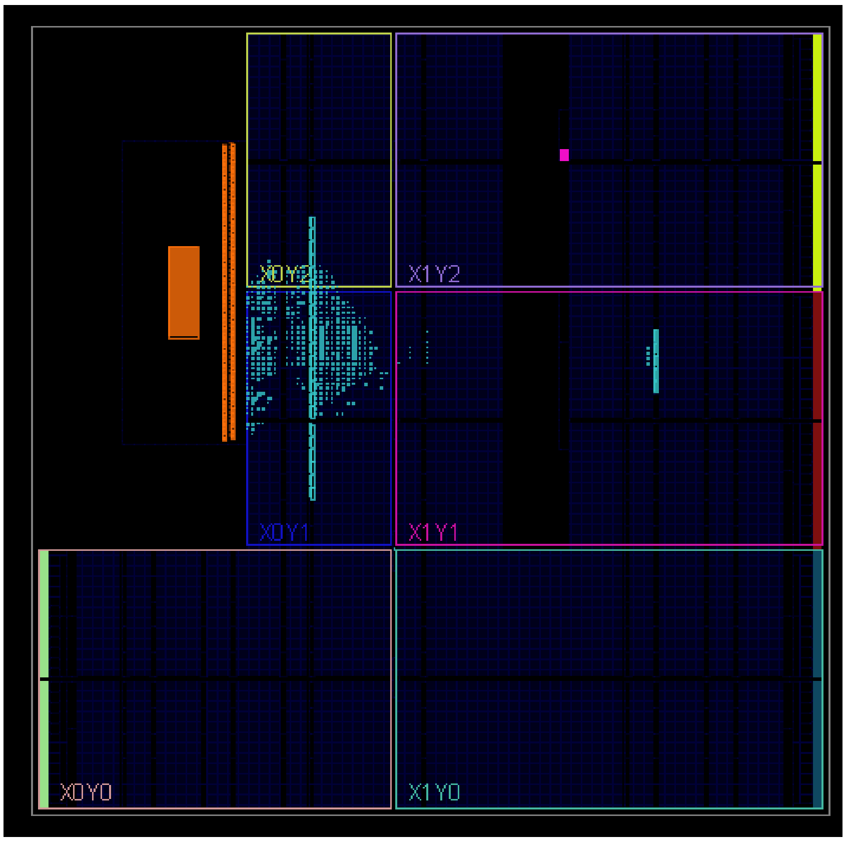


Figure 34. Visualization of utilized resources in the FPGA for mat_calc overlay.

4. Conclusion, discussion and future work

The purpose of the project was to understand the feasibility of using a PYNQ board for real time 3D model generation. Through the project it was realized that there are a lot of bottlenecks in the whole pipeline from 3D point cloud generation to rendering the reconstructed 3D object on a 2D plane for visualisation. Since optimising the full pipeline was outside the scope of the project, different sections were run using different processing units. The stereo video input was simulated by splitting the mono camera. Then to generate more accurate depth maps, image dataset was used containing aloe vera images. A custom 3D to 2D projection algorithm was employed to visualize the model as existing tools like mesh lab would be very intense for real time application. To prove that PYNQ board can be used for acceleration, a custom overlay was created for this projection. The overlay was successfully able to replicate the original algorithm but can be accelerated much further. In future work all the bottlenecks mentioned throughout the project would be accelerated by creating custom overlays, such as for instance, overlays for disparity map and 3D point cloud generation. Such designs would enable 3D scene reconstruction and visualisation in real time.

This work proves that both the portable and parallel computation capacity of PYNQ board makes them the ideal candidate for real-time 3D modelling. The authors hope that this would inspire more developers to work on practical applications such as drone mapping or GoPro for on the go photogrammetry. The development platform makes it very easy to prototype such applications and the future is full of interesting possibilities with PYNQ.

Some of the next steps for this project would be to begin looking at data handling on the board to make full use of the accelerated hardware multiplications. This would also allow for further parallelisation of the rotation and projection calculations either by performing multiple operations at a time or processing multiple points at a time. Another step would be to create a complete pipeline for this process from video acquisition through to output projection of 3D space. Other steps that can be taken are to implement accelerators for the calculation of the point cloud or the disparity/depth map. All of these steps would push this project closer to a complete real time 3D render and visualisation of an object or environment.

References

[1] OpenCV, "Camera calibration With OpenCV," 31 December 2019. [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html .

[2] D. Falkingham, "Photogrammetry testing 14: AliceVision Meshroom," [Online]. Available: https://peterfalkingham.com/2018/08/11/photogrammery-testing-14-alicevision-meshroom/. . [Accessed 10 July 2020].

[3] R. T. J. R. I Kuon, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation,* vol. 2, no. 2, pp. 135-253, 2007.

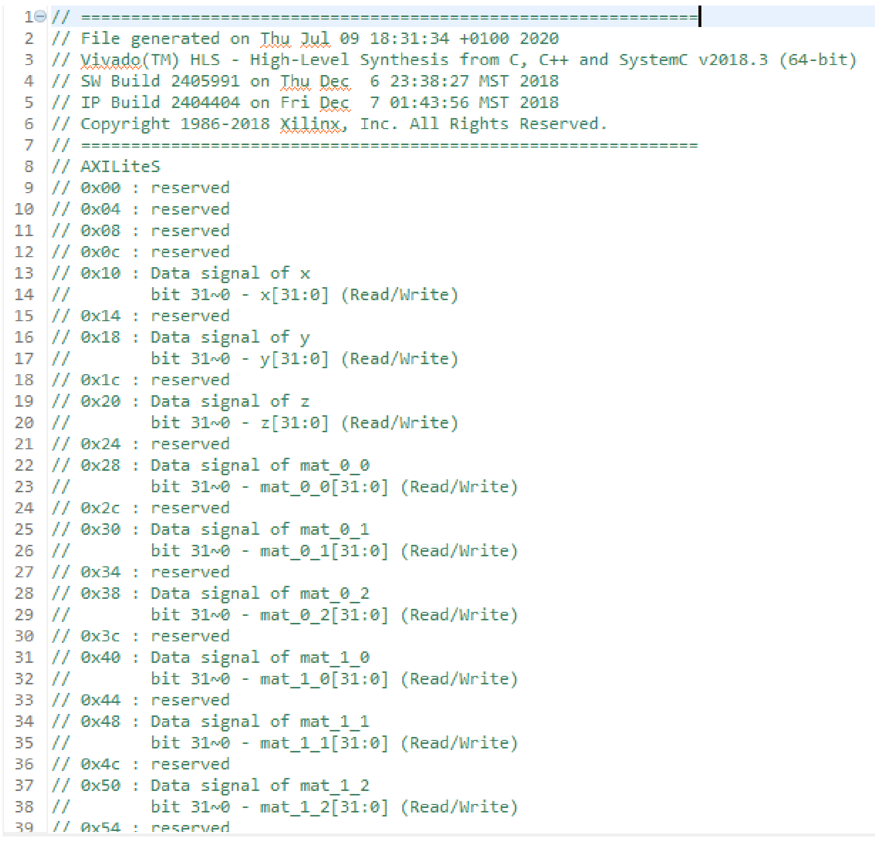
[4] L. D. R. S. J. &. W. J. Alvarez, "Dense disparity map estimation respecting image discontinuities: A PDE and scale-space based approach," *Journal of Visual Communication and Image Representation,* vol. 13, no. 1-2, pp. 3-21, 2002.

[5] A. Mordvintsev, "Depth Map from Stereo Images — OpenCV-Python Tutorials 1 documentation", Opencv-python-tutroals.readthedocs.io," [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_depthmap/py_depthmap.html. . [Accessed 10 July 2020].

Appendix

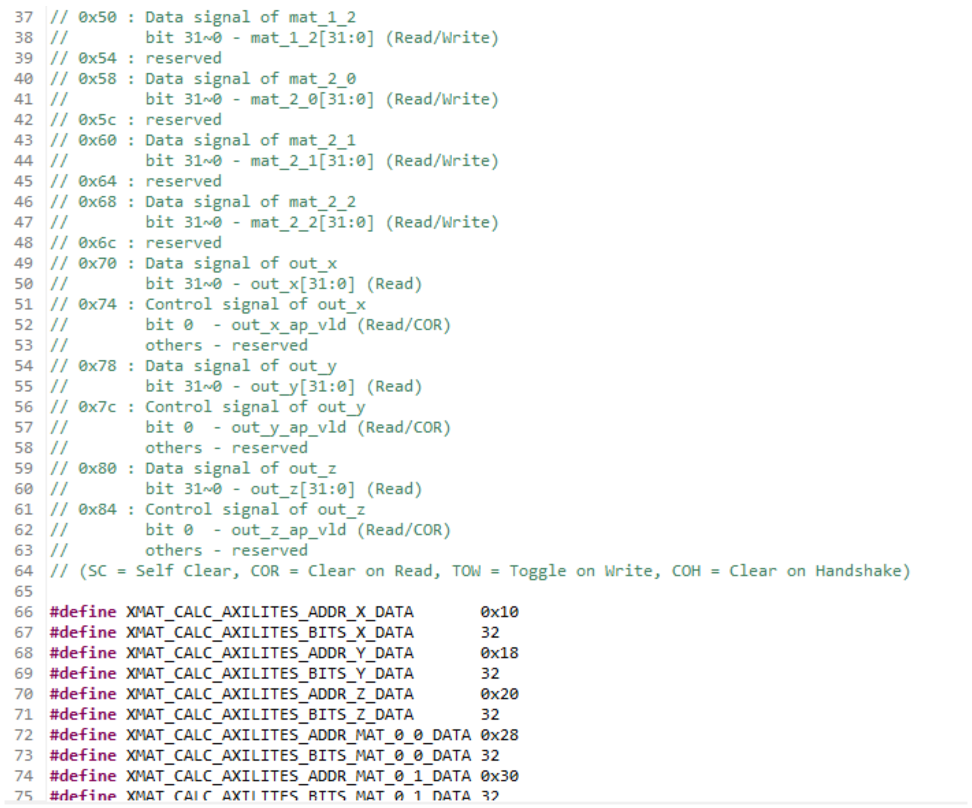
The GitHub repository can be found in the following link: https://github.com/smpis/PyPix

The ports generated in Vivado HLS for the mat_calc overlay are illustrated in the Figures below.



```
// ==============================================================
// File generated on Thu Jul 09 18:31:34 +0100 2020
// Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC v2018.3 (64-bit)
// SW Build 2405991 on Thu Dec  6 23:38:27 MST 2018
// IP Build 2404404 on Fri Dec  7 01:43:56 MST 2018
// Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
// ==============================================================
// AXILiteS
// 0x00 : reserved
// 0x04 : reserved
// 0x08 : reserved
// 0x0c : reserved
// 0x10 : Data signal of x
//        bit 31~0 - x[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of y
//        bit 31~0 - y[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of z
//        bit 31~0 - z[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of mat_0_0
//        bit 31~0 - mat_0_0[31:0] (Read/Write)
// 0x2c : reserved
// 0x30 : Data signal of mat_0_1
//        bit 31~0 - mat_0_1[31:0] (Read/Write)
// 0x34 : reserved
// 0x38 : Data signal of mat_0_2
//        bit 31~0 - mat_0_2[31:0] (Read/Write)
// 0x3c : reserved
// 0x40 : Data signal of mat_1_0
//        bit 31~0 - mat_1_0[31:0] (Read/Write)
// 0x44 : reserved
// 0x48 : Data signal of mat_1_1
//        bit 31~0 - mat_1_1[31:0] (Read/Write)
// 0x4c : reserved
// 0x50 : Data signal of mat_1_2
//        bit 31~0 - mat_1_2[31:0] (Read/Write)
// 0x54 : reserved
```

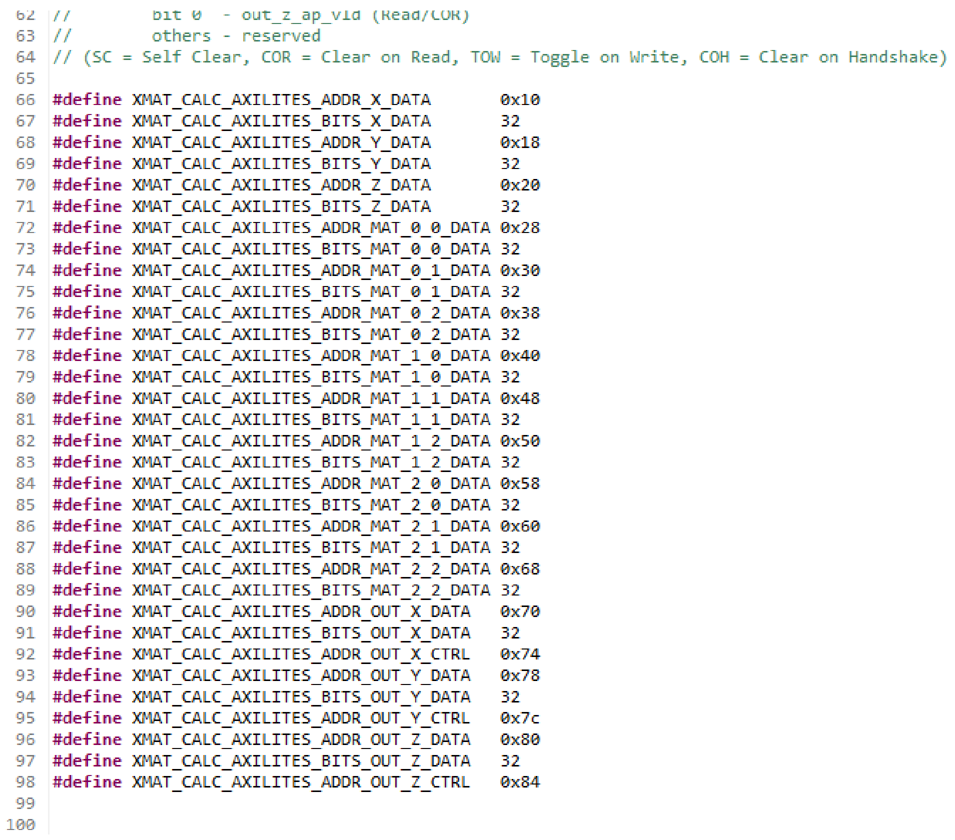
Figure 35. AXI interface ports, part 1.



```
// 0x50 : Data signal of mat_1_2
//        bit 31~0 - mat_1_2[31:0] (Read/Write)
// 0x54 : reserved
// 0x58 : Data signal of mat_2_0
//        bit 31~0 - mat_2_0[31:0] (Read/Write)
// 0x5c : reserved
// 0x60 : Data signal of mat_2_1
//        bit 31~0 - mat_2_1[31:0] (Read/Write)
// 0x64 : reserved
// 0x68 : Data signal of mat_2_2
//        bit 31~0 - mat_2_2[31:0] (Read/Write)
// 0x6c : reserved
// 0x70 : Data signal of out_x
//        bit 31~0 - out_x[31:0] (Read)
// 0x74 : Control signal of out_x
//        bit 0  - out_x_ap_vld (Read/COR)
//        others - reserved
// 0x78 : Data signal of out_y
//        bit 31~0 - out_y[31:0] (Read)
// 0x7c : Control signal of out_y
//        bit 0  - out_y_ap_vld (Read/COR)
//        others - reserved
// 0x80 : Data signal of out_z
//        bit 31~0 - out_z[31:0] (Read)
// 0x84 : Control signal of out_z
//        bit 0  - out_z_ap_vld (Read/COR)
//        others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XMAT_CALC_AXILITES_ADDR_X_DATA       0x10
#define XMAT_CALC_AXILITES_BITS_X_DATA       32
#define XMAT_CALC_AXILITES_ADDR_Y_DATA       0x18
#define XMAT_CALC_AXILITES_BITS_Y_DATA       32
#define XMAT_CALC_AXILITES_ADDR_Z_DATA       0x20
#define XMAT_CALC_AXILITES_BITS_Z_DATA       32
#define XMAT_CALC_AXILITES_ADDR_MAT_0_0_DATA 0x28
#define XMAT_CALC_AXILITES_BITS_MAT_0_0_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_0_1_DATA 0x30
#define XMAT_CALC_AXILITES_BITS_MAT_0_1_DATA 32
```

Figure 36. AXI interface ports, part 2.



```
//        bit 0  - out_z_ap_vld (Read/COR)
//        others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XMAT_CALC_AXILITES_ADDR_X_DATA       0x10
#define XMAT_CALC_AXILITES_BITS_X_DATA       32
#define XMAT_CALC_AXILITES_ADDR_Y_DATA       0x18
#define XMAT_CALC_AXILITES_BITS_Y_DATA       32
#define XMAT_CALC_AXILITES_ADDR_Z_DATA       0x20
#define XMAT_CALC_AXILITES_BITS_Z_DATA       32
#define XMAT_CALC_AXILITES_ADDR_MAT_0_0_DATA 0x28
#define XMAT_CALC_AXILITES_BITS_MAT_0_0_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_0_1_DATA 0x30
#define XMAT_CALC_AXILITES_BITS_MAT_0_1_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_0_2_DATA 0x38
#define XMAT_CALC_AXILITES_BITS_MAT_0_2_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_1_0_DATA 0x40
#define XMAT_CALC_AXILITES_BITS_MAT_1_0_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_1_1_DATA 0x48
#define XMAT_CALC_AXILITES_BITS_MAT_1_1_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_1_2_DATA 0x50
#define XMAT_CALC_AXILITES_BITS_MAT_1_2_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_2_0_DATA 0x58
#define XMAT_CALC_AXILITES_BITS_MAT_2_0_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_2_1_DATA 0x60
#define XMAT_CALC_AXILITES_BITS_MAT_2_1_DATA 32
#define XMAT_CALC_AXILITES_ADDR_MAT_2_2_DATA 0x68
#define XMAT_CALC_AXILITES_BITS_MAT_2_2_DATA 32
#define XMAT_CALC_AXILITES_ADDR_OUT_X_DATA   0x70
#define XMAT_CALC_AXILITES_BITS_OUT_X_DATA   32
#define XMAT_CALC_AXILITES_ADDR_OUT_X_CTRL   0x74
#define XMAT_CALC_AXILITES_ADDR_OUT_Y_DATA   0x78
#define XMAT_CALC_AXILITES_BITS_OUT_Y_DATA   32
#define XMAT_CALC_AXILITES_ADDR_OUT_Y_CTRL   0x7c
#define XMAT_CALC_AXILITES_ADDR_OUT_Z_DATA   0x80
#define XMAT_CALC_AXILITES_BITS_OUT_Z_DATA   32
#define XMAT_CALC_AXILITES_ADDR_OUT_Z_CTRL   0x84
```

Figure 37. AXI interface ports, part 3.