Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Abdelrahman Mohamed

High-Performance Embedded Systems for Stereoscopic Vision

Master's thesis in Embedded Computing Systems
Supervisor: Prof. Geir Mathisen, Dr. Kristoffer Nyborg Gregertsen
June 2019

NTNU
Norwegian University of Science and Technology

Abdelrahman Mohamed

High-Performance Embedded Systems for Stereoscopic Vision

Master's thesis in Embedded Computing Systems
Supervisor: Prof. Geir Mathisen, Dr. Kristoffer Nyborg Gregertsen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Preface

This thesis "High-Performance Embedded Systems for Stereo Vision" has been written as a part of the European Master's Course in Embedded Computing Systems (EMECS), fulfilling a master's degree in Embedded Computing Systems at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisors Prof. Geir Mathisen and Dr Kristoffer Nyborg Gregertsen for their excellent guidance and support during this challenging project, and providing me with all the tools required for this thesis, Finally I would like to thank my family and friends for being helpful and supportive during the time of my studies.

The stereo system developed for this project is built for Zynq Ultrascale+, and Nvidia Jetson boards, and it utilizes the libraries OpenCV, CUDA and XFopenCV, more details can be found in the methods and tools section 4.

The project was undertaken at Sintef Digital, where I had previously worked on my specialization project, however the contents of this thesis are original, independent work by the author, Abdelrahman Mohamed.

Abdelrahman Mohamed
Trondheim, 7 July 2019

Abstract

This thesis aims to extract real-time 3D information about the environment using stereoscopic vision in space; it was developed for the Integrated 3D Sensors project I3DS (Integrated 3D sensors).
Stereo vision presents two significant challenges first is the correspondence problem; finding for every pixel it's corresponding pixel in the other image, two approaches for solving this problem are explored in this thesis local-based approach and semi-global based approach. The second challenge is the high throughput of data, making it challenging to implement a system that provides high frame rates with low power consumption, different embedded architectures are explored in this thesis; embedded CPU, embedded GPU, and FPGA to mitigate that problem.
The use of hardware accelerators (FPGA and GPU) provides better performance but at a significantly higher development costs, and time, so this project exploits modern software tools to accelerate the embedded development, OpenCV on the embedded CPU, CUDA implementation of OpenCV and libsgm on the embedded GPU, and the new library by Xilinx XFopenCV synthesized for the FPGA.
Finally, local block stereo matching and semi-global stereo matching are implemented on CPU, GPU and FPGA, providing six different implementations, benchmarked for stereo matching accuracy, processing time/frame rates and power consumption.
It is shown that the CPU implementation provides the highest accuracy, but lowest frame rates, and highest power consumption, the FPGA implementation provided lowest accuracy, but highest frame rates and lowest power consumption, while the GPU scored mid-way in all of its benchmarks.
This thesis starts with an introduction and then provides the theoretical background behind the implementations in the High-Performance Embedded Computing and Stereo Vision chapters, afterwards the implementations, and their optimizations are documented in the Stereo system calibration and the embedded stereo system implementation chapters, finally the benchmarks and results chapter followed by the conclusion and discussion.

Contents

List of Figures

1 Introduction

Space robotics is an exciting field with many applications both in low-earth orbit and planetary exploration, such as active debris removal [10] and Mars exploration [11] simultaneous localization and mapping. However, many challenges are faced with the ability to map the environment and localize objects of interests, the "robot" not only needs to be able to understand the contents of 2D images, but also be able to perceive the depth information. There are many sensors that provide 3D information of the environment, one of the most commonly used are stereo cameras.

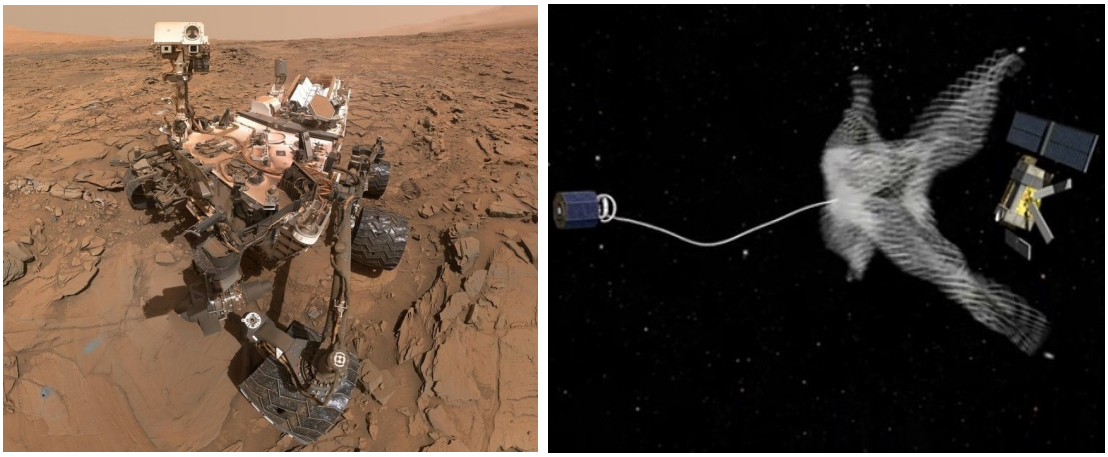


Figure 1: Image on the left Curiousity Mars Rover [Credit:NASA/JPL-Caltech/MSSS] Image on the right active debris removal in low earth orbit [Credit: ESA]

Stereo cameras, as inspired from nature, takes two images from two different viewpoints at the same time instance, from these images the 3D information of the environment can be extrapolated, it has multiple applications in embedded systems from autonomous cars [12], space [13] and robotics [14] [15]. Stereo vision remains an active area of research, especially with finding how every pixel in a left image correspond to which pixel in the right image; the stereo correspondence is the most challenging aspect [16] within the stereo pipeline, and this project explores different methods to solve this problem.

The use of a typical embedded microprocessor architecture fails to meet the high data throughput demands of stereo-vision algorithms while keeping power consumption to a minimum. Conventional space-grade processors such as LEON3 and RAD750 perform significantly worse than the commercial off the shelf devices [13], which gives the need for high-performance embedded computing [1], employing different architectures to solve different parts within the problem (Heterogeneous Computing Architecture) [17], and using hardware accelerators such as Field

Programmable Gate Arrays (FPGAs) [18] and Graphical Processing Units (GPUs) [19].

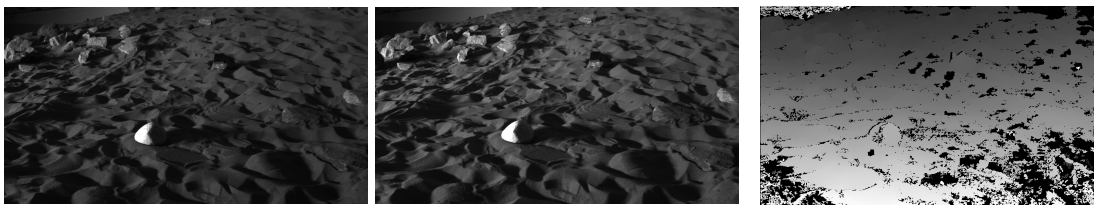


Figure 2: Stereo image pair from the planetary I3DS data-set, figure on the right is the disparity map generated using a semi-global block matching algorithm

Stereo vision is an "embarrassingly parallel" problem which means the problem requires little to no effort, to be separated into smaller parallel problems which make hardware accelerators (FPGAs and GPUs) work exceptionally well for accelerating the stereo pipeline, due to their highly parallel architecture

This thesis considers the use of high performance embedded systems, for stereoscopic vision in space, it was developed for the Integrated 3D Sensors project I3DS (Integrated 3D sensors) which is funded under Horizon 2020 EU research and development program and is part of the Strategic Research Cluster on Space Robotics Technologies. The I3DS project is developing a generic modular sensors suite, which includes various sensors which are to be used for near-future space exploration missions, and solving the challenges of integrated pre-processing and integrating it with the electrical, thermal and mechanical interfaces of the vehicle.

2 High-Performance Embedded Computing

For as long as embedded systems have existed, there has been a need for high performance and lower energy consumption; Over the last three decades, embedded systems performance has been improving almost exponentially.
One of the main drivers for such developments was frequency scaling up until the mid-2000s; afterwards multiple technologies have been employed to meet the need for higher performance and lower energy consumption, such as pipelining and multicore architectures. However, now one of the main techniques used when the non-functional requirements are not met and more code optimizations is no longer feasible, Sections of the code can be moved to hardware accelerators to achieve the performance needs, such as embedded graphical processing units (GPU) and field programmable gate array (FPGA).

2.1 Graphical Processing Units

Graphical Processing Units primary purpose was to accelerate the graphical pipeline. However, it can now be used to accelerate many computationally intensive tasks. GPUs offer a massively parallel architecture, with the use of Single Instruction Multiple Data (SIMD) it offers a massive boost for the acceleration of "embarrassingly parallel tasks" such as the stereo block matching algorithm. Embedded GPUs are optimized to run with low power, figure 3 shows the architecture of a Mali GPU with multiple cores accessing shared internal memory.

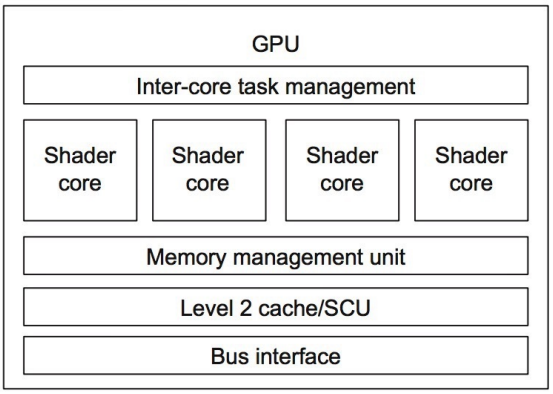


Figure 3: Block diagram of the Mali GPU microarchitecture [1]

GPUs offer some advantages over other types of hardware accelerators most significant advantage is that it runs multiple floating-point operations in parallel, for example; the Nvidia Jetson

TX2 offers 256 cores allowing up to 256 floating-point operations to be running in parallel. Another advantage is that the algorithm is developed in software, incurring lower development costs and time, better backwards compatibility and upgradability. However, this comes at a cost; instructions have to be fetched and cued up, math operations have to be performed, and results have to be sent to memory.

2.1.1 Streaming multiprocessors

The GPU is built around an array of streaming multiprocessors, figure 4 shows the Pascal architecture with multiple cores; 64 floating point single precision cores and 32 floating point double precision cores, LD/ST are load/store units they enable overlapping load/store instructions, and SFU is a special function unit.

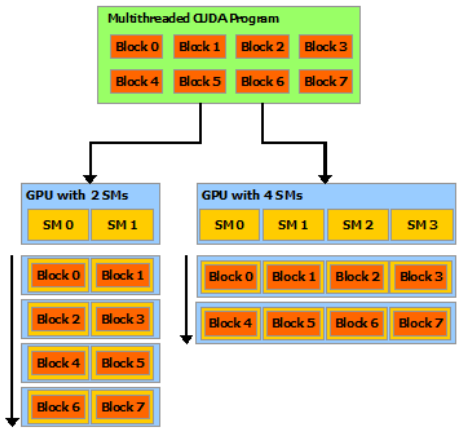
Figure 4: Block diagram of a streaming multiprocessor unit in the Pascal architecture [2]

2.1.2 CUDA

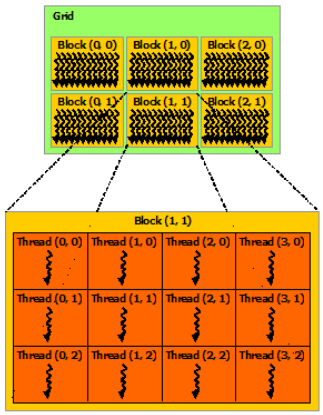
CUDA (Compute Unified Device Architecture) is a parallel computing platfrom developed by NVIDA, it enables the development of applications that leverages the GPU's high levels of parallelism,and was designed to be automatically scalable with the number of streaming multiprocessors and memory partitions as shown in figure 5a.

The functions set to run on hardware invoke a kernel, creating a thread hierarchy that can be grouped into thread blocks as shown in figure 5b, for the image processing application; these blocks are configured into a 2-dimensional grid of blocks, which is also configured into a 2-dimensional grid of threads. The grid size depends on the image size, and can be configured so that there are as many threads as there are pixels.

The GPU employs shared memory to accelerate synchronization for the image processing application, The blocks can store multiple pixels data in the shared memory, giving each thread the ability to access the surrounding pixels of it's pixel.



(a) Automatic scalability in a CUDA program (Reference:[20])



(b) Typical CUDA grid of blocks (Reference:[20])

2.2 Field Programmable Gate Array

Field programmable gate arrays (FPGA) are inherently highly flexible and reconfigurable allowing the implementation of customized hardware including customized memory, control units and datapath, thus the developer can design efficient hardware to match the application, for example, applications such as image processing, parallel pixel level operations can be employed to providing a system well suited for real-time applications.

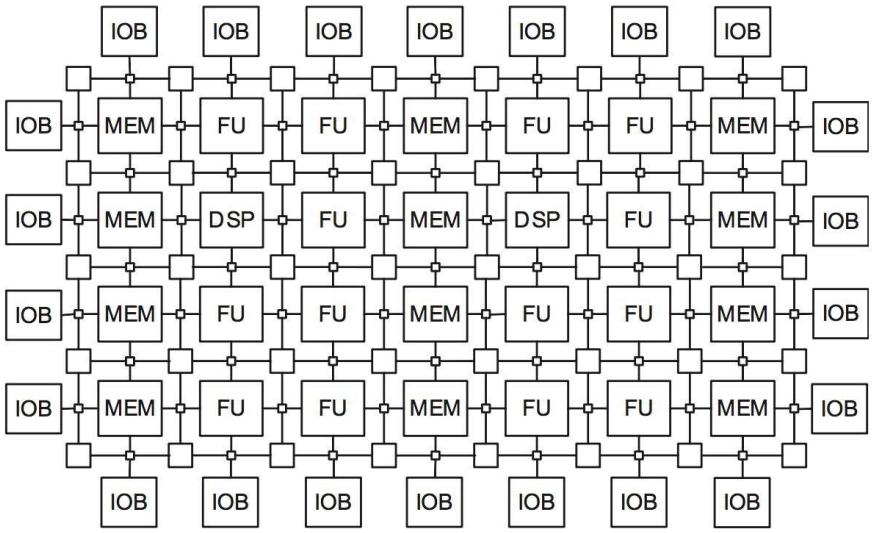


Figure 6: Example architecture of reconfigurable hardware [1]

The figure 13 illustrates example hardware within the FPGA, the programmable logic consists of input output blocks (IOB), configurable logic blocks (CLB), digital signal processing components (DSP), Memory components (MEM) and interconnect.

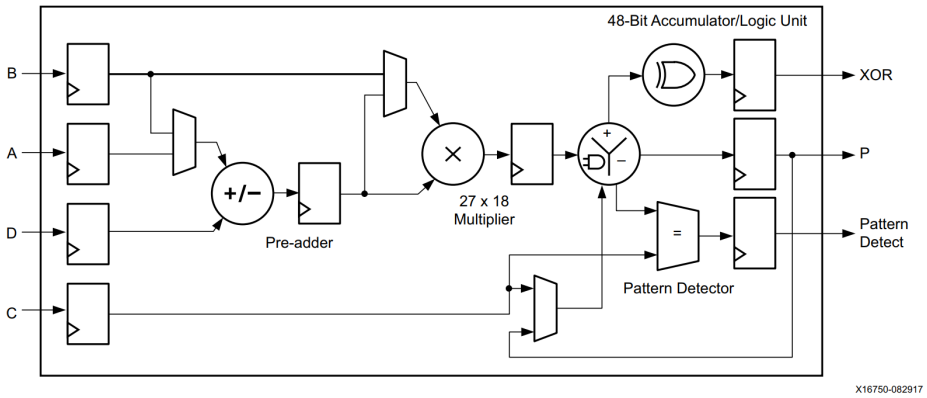


Figure 7: The built-in components in the XtremeDSP DSP48 slices that are in the Zynq Ultrascale+ architecture [3]

When a function is chosen to be accelerated on hardware, it can be configured as in figure 8 with multiple computing engines, a suitable communication structure is chosen (e.g. RAM, FIFO), taking advantage of such flexiblity allows for designs higher performance and lower power consumption.

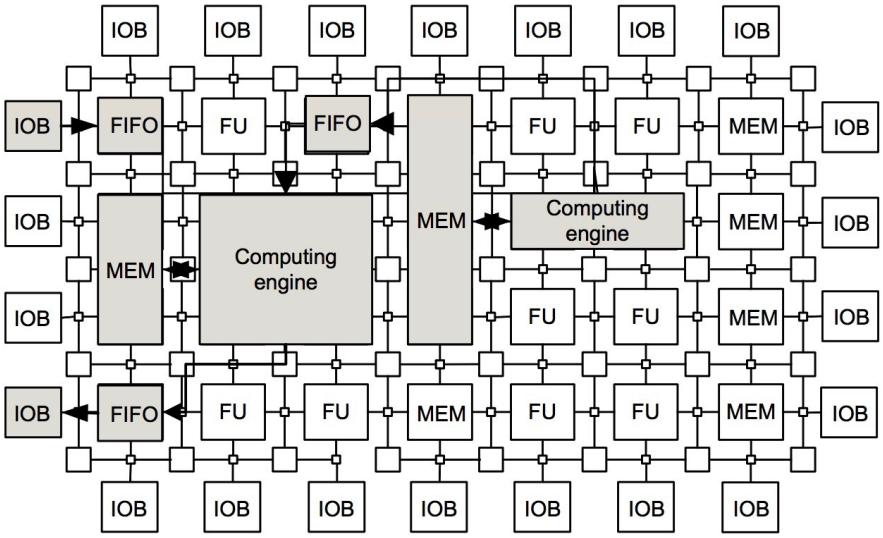


Figure 8: Example accelerator implemented on reconfigurable hardware [1]

It's difficult to deliver a hard real-time system on multicores running software, due to pro-

	rad-hard CPU (1-core)	embedded CPU (1-core)	desktop CPU (1-core)	mobile GPU	high-end DSP	FPGA	desktop GPU
Throughput	0.2−1.7	0.5−2	20−100	50−150	50−240	300−1460	200−2000
Power (W)	1−18	1−2	20−90	6−10	1−10	2−10	70−195
Perf/Watt	0.1−0.6	0.25−2	0.5−1	8−15	12−50	60−250	5−25

Figure 9: Comparison of different processing platforms [4]

cesses interrupting each other, worst-case execution time, constraints for the criticality of the process and safety. In comparison, FPGAs software analysis for FPGAs is less complex but require increased development time. SoC approach is the most promising for high-performance embedded computing, combining acceleration resources with general purpose processors and peripherals lead to architectures with single-device, low-power, small size/weight and fast intra-chip communication to support efficient co-processing.

2.3 Comparison of architectures for vision applications in space

High-performance embedded systems have become increasingly important for applications in space such as planetary exploration and active debris removal. High-performance embedded systems are with multiple challenges such as performance to able to process a high throughput of data in real-time, As the system is mainly powered through solar energy, energy efficiency is critical, mass and volume also pose a challenge as they induce high costs.
However, a unique challenge that comes to light for space applications is the radiation tolerance, which is covered by rad-hard and rad-tolerant devices that guarantee reliability but offer much lower performance compared to the commercial off the shelf devices.
The use of high-performance embedded systems in space would provide significant improvement for vision applications in space, as shown by Lentaris et al. [4]. The mobile GPU shows an order of magnitude higher performance per watt, while the FPGA shows two orders of magnitude higher performance per watt than the rad-hard CPU and the embedded CPU.

2.4 Heterogeneous computing

Computer vision algorithms can be quite complex, different functions can run more efficiently on different architectures, if the system was designed to run on a single target, it often doesn't meet the performance, and power budget constraints, so heterogeneous architectures pair up different architectures on the same chip providing the ability to exploit every architecture. However, it comes with challenges.

- Choosing which functional blocks to run on which hardware
- The data flow between different processing systems need to be synchronized while satisfying the timing constraints
- Different architectures can have different programming environments and languages

Employing multiple architectures within a single chipset has become more common in recent years, the two architectures that are used for this thesis are good examples for that. First one produced by Xilinx; the Zynq Ultrascale+ EG architecture 10 which includes an application processing unity (ARM Cortex A53), realtime processing unit (ARM Cortex R5), graphics processing unit (ARM Mali 400) and the programmable logic. The second architecture produced by NVIDIA, The tegra X2 architecture 11 used in the Nvidia Jetson TX2, which has Pascal architecture GPU with 256 CUDA cores, Denver 2 dual core CPU and Cortex A57 quad core CPU.

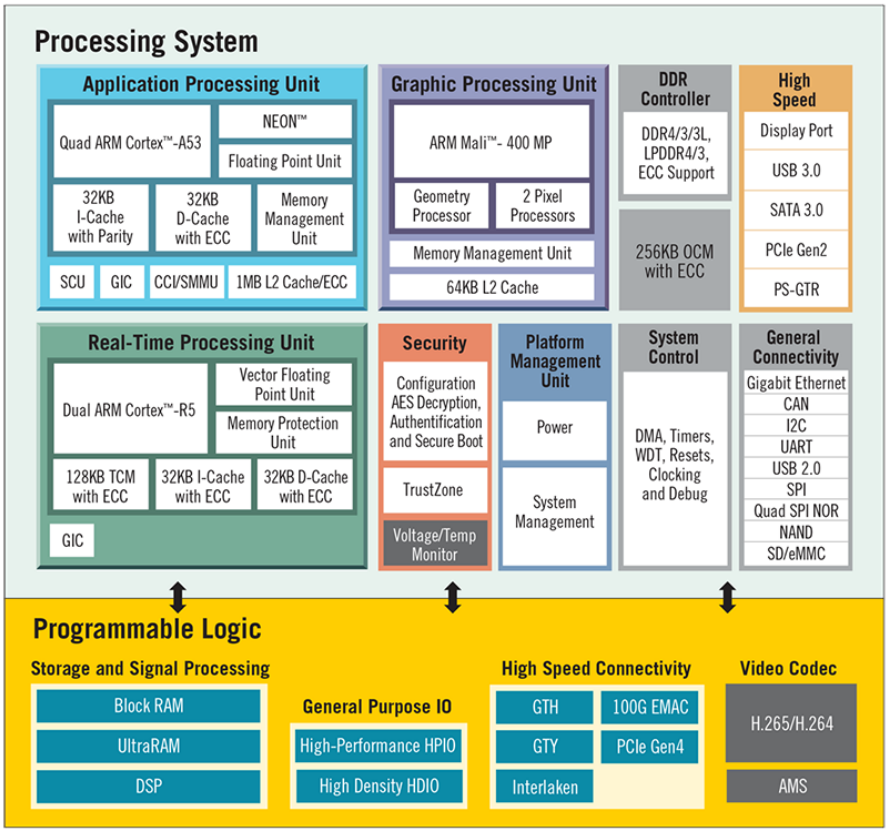


Figure 10: Zynq Ultrascale+ EG architecture

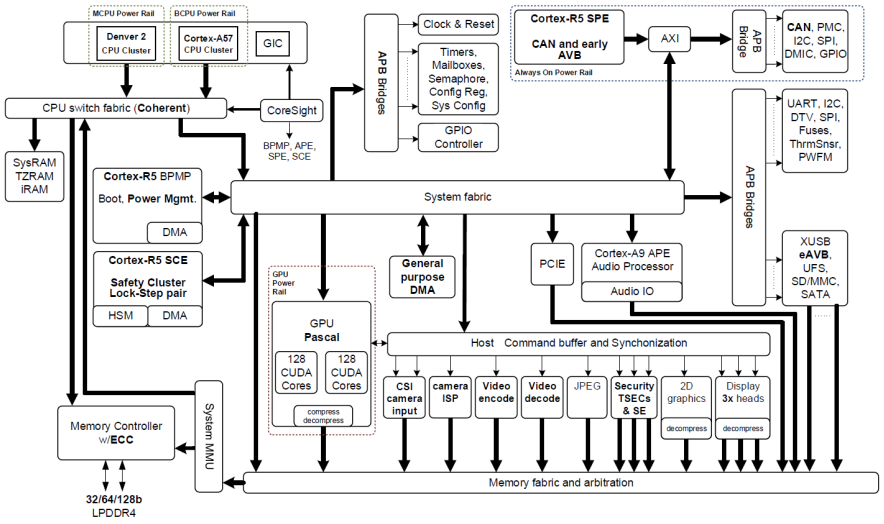


Figure 11: Tegra X2 architecture

3 Stereo Vision

Stereo vision is a sensing technique that uses multiple cameras to estimate the 3D position of objects from two cameras in two different positions. The object is detected in both cameras in different positions within the image, the difference between the positions is called disparity, from the disparity, the depth information is calculated.
Stereo vision has been researched heavily over the past 3 decades, with applications spanning from self-driving cars, space exploration to medical imaging. However, it is still faced with many challenges.
The correspondence problem is a difficult challenge [16] and there are multiple algorithms designed to solve it [21] finding which pixel in an image correlates with which pixels in the other image. Another challenge is the real-time performance of the stereo system; stereo vision handles a high throughput of data coming from two cameras. Due to the embedded CPU architecture, it fails to deliver real-time performance in a power constrained application. Architectures such as embedded GPUs and FPGAs can solve this problem, but an algorithm that takes full advantage of the highly parallel nature of both processing systems need to be implemented.

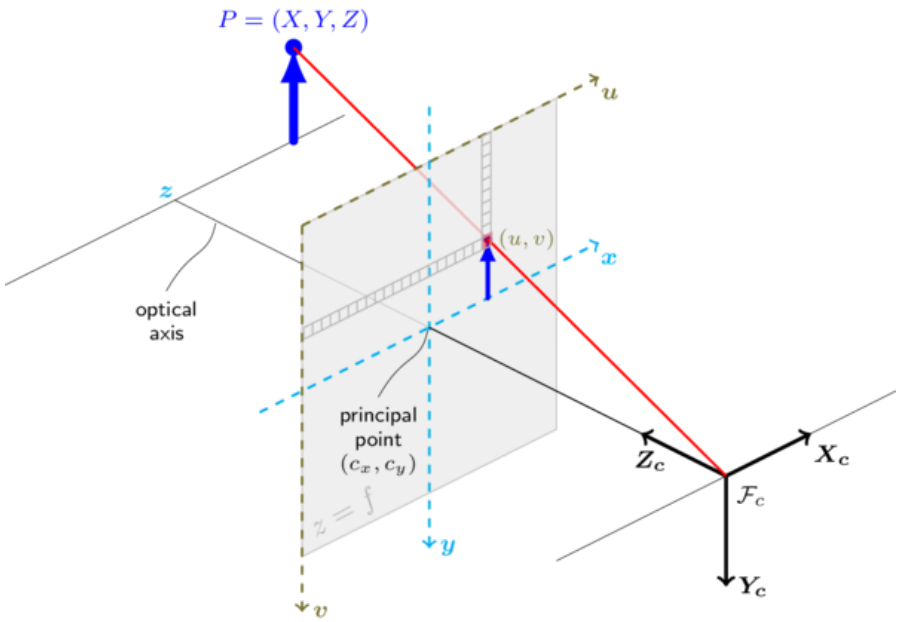


Figure 12: The pinhole camera model. The camera has an origin point C and the image forms as pixel p as projection of points P at an offset f [5]

3.1 Stereo system geometry

Estimating the depth information of the scene requires full knowledge of the stereo vision system, each camera's optical characteristics and position/orientation need to be modelled to provide accurate results.

3.1.1 Pinhole camera model

The pinhole camera model 12 provides a relationship of a three-dimensional point $P = (X, Y, Z)$ with a pixel position $p = (x, y)$; however, this model assumes no distortion from a lens and assumes the aperture as a point. The pixel $p = (x, y)$ is the projection of the 3D point $P = (X, Y, Z)$ on an image plane that is at an offset f focal length from the camera origin C resulting in the relationship 3.1.

$$x = f\frac{X}{Z} \qquad y = f\frac{Y}{Z} \tag{3.1}$$

Every 3D point P uniquely corresponds to a pixel p, however, for every pixel the corresponding point can lie anywhere along the red line, so to retrieve depth information from pixel positions a second camera is required.

3.1.2 Depth computation

depth data can be calculated to produce a depth map of a scene from the positions of the corresponding pixels in each image by using triangulation.

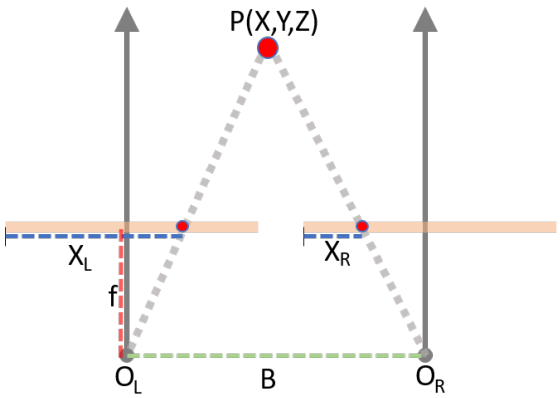


Figure 13: Depth calculation from two perfectly aligned pinhole cameras

The difference between the corrosponding pixels in both images is called disparity d, and in the case of perfectly aligned pinhole cameras lies on the x axis of the image, so we can calculate the disparity by simply subtracting the x position of each pixel X_L and X_R

$$d = X_L - X_R \tag{3.2}$$

Once the disparity information is calculated, the depth information can be extracted from the disparity d, focal length f and the distance between both cameras (Baseline b).

$$Z = f\frac{b}{d} \tag{3.3}$$

3.1.3 Epipolar geometry

The epipolar geometry describes the relationship between both cameras in the stereo camera and can be used to build it's model.

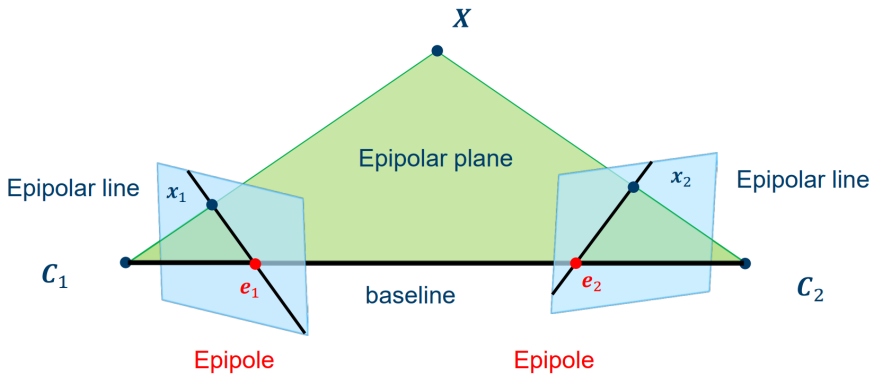


Figure 14: Illustration of the epipolar geometry [6]

The epipolar plane is the plane containing X, and the two camera centres C_1 and C_2, the baseline is the line joining the two camera centres, the epipolar lines are where the epipolar plane intersects the image planes, the epipoles are where the baseline intersects the two image planes.

Epipolar constraint

With knowledge about the epipolar geometry, the stereo matching algorithm can be made more efficient, by constraining the search for correlating pixels along one dimension; the epipolar line.

3.2 Stereo camera model

The stereo image is said to be fully rectified, if the epipolar lines lie horizontaly within the images, effictively making the search for the corresponding pixels, on pixels with the same y position.
To making rectification possible, the stereo pair need to be rectified in real-time, this is achieved by modeling the camera's optical parameters (intrinsic parameters), and their position/orientation (extrinsic parameters).

3.2.1 Intrinsic parameters

The previous relationship is expressed in homogeneous coordinates 3.4.

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{I_{3x3}} | \mathbf{0_{3x1}} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (3.4)$$

Where λ is a scaling term used to keep the third term normalized to 1, (x, y) is the pixel position, of the 3D point (X, Y, Z), the intrinsic paramaters of the camera are represented in the camera

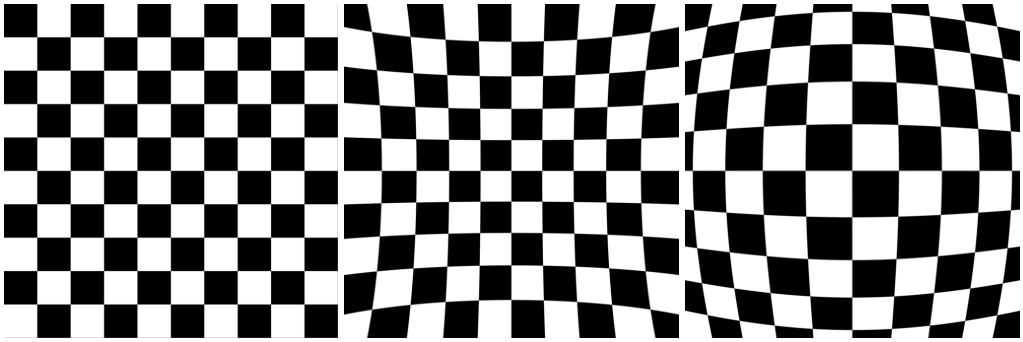


Figure 15: A checkerboard pattern demonstrated with no distortion, positive radial distortion and negative radial distortion respectively

matrix **A**.

$$\mathbf{A} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

The focal length f is split into two components f_x and f_y, c_x and c_y are the principal point's position relative to the reference frame, and the pixel skew effect s which can usually be ignored for modern cameras, saving up computation time in a resource constraint system.

3.2.2 Extrinsic parameters

The camera itself can move around if it is attached to a dynamic object (robot), so describing the position of the camera can be achieved by moving from the camera coordinate system to the world coordinate system.
The position and orientation of each camera relative to each other and to the world coordinate system are defined in the extrinsic parameters as transformation matrices 3.6.

$$[\mathbf{R}_{3x3}|\mathbf{T}_{3x1}] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \tag{3.6}$$

Finally, the pixel positions in terms of the 3D point can be calculated 3.7, while taking into consideration the intrinsic and extrinsic parameters.

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.7}$$

3.2.3 Distortion

The biggest source of distortion, however, comes from the lens of the camera, which applies radial distortions 15 and some tangential distortion, which makes the image appear more warped. Taking into consideration the distortion coming from the lenses; the camera model can be ex-

tended using the extrinsic parameters from 3.6.

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{3x3} | \mathbf{T}_{3x1} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.8}$$

$$r^2 = x'^2 + y'^2 \tag{3.9}$$

$$\begin{aligned} x'' &= x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y'' &= y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_2 x' y' + p_1(r^2 + 2y'^2) \end{aligned} \tag{3.10}$$

where k_1,k_2 and k_3 are the radial distortion coefficients and p_1, p_2 are the tangential coefficients

3.3 Image rectification

The calibration process will extract the intrinsic, extrinsic and distortion parameters. The images from the camera will then need be rectified in real-time using these parameters, proving fully rectified images to the stereo matching algorithm.

There are many rectification algorithms each having it's own advantage, since the thesis is concerned with the implementation on power constrained highly-parallel embedded system architectecture, a well suited algorithm from Zicari et al. [22] can be used to rectify and undistort the images, highlighted in the following series of equations.

$$a_1 = 2.x.y \qquad a_2 = r_2 + 2.x^2 \qquad a3 = r_2 + 2.y_2 \tag{3.11}$$

$$r_2 = x^2 + y^2 \qquad r_4 = {r_2}^2 \qquad r_6 = {r_2}^3 \tag{3.12}$$

$$\begin{bmatrix} x_d \\ y_d \end{bmatrix} = (1 + k(1).r_2 + k(2).r_4 + k(5).r_6). \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} k(3).a_1 + k(4).a_2 \\ k(3).a_3 + k(4).a_1 \end{bmatrix} \tag{3.13}$$

$$\begin{bmatrix} x_{raw} \\ y_{raw} \\ 1 \end{bmatrix} = \mathbf{A} \times \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} \tag{3.14}$$

Where A is the camera matrix 3.5, extended with the extrinsic transformations 3.7 and K is the distortion vector.

These equations taken in a pixel location $p(x, y)$ and output a raw rectified and undistorted new raw pixel values $p_{raw}(x_{raw}, y_{raw})$ which is non integer.

In order to calculate the new rectified pixel position p_{rect} interpolation is employed; x_{raw} can be split into it's integer component x_i and it's non integer component x_f which leads to $P_{rect} = (x_{rect}, y_{rect})$ can be computed by interpolating the four surrounding pixels.

$$\begin{aligned} p_{rect} = (1 - x_f)(1 - y_f)p(x_i, y_i) + (1 - x_f)y_f.p(x_i, y_{i+1}) \\ + x_f(1 - y_f)p(x_{i+1}, y_i) + x_f.y_f.p(x_{i+1}, y_{i+1}) \end{aligned} \tag{3.15}$$

From 3.15 the rectified pixel value of every pixel in the image can be calculated, producing a rectified image will be used as input for the stereo matching block.

3.4 Stereo matching

The process of taking two images from different views, and outputting the depth information for every pixel is called stereo matching, The three dimensional position of every pixel can be known from the position difference between corresponding pixels (Disparity), which is inversely proportional to the depth position.

3.4.1 The correspondence Problem

Finding which pixel within the other image corresponds to every pixel in an image has proven to be the most challenging step. However this process can be made less computationally demanding by reducing the search area of the corresponding pixels, one method is that if the input images at this stage are fully rectified, the corresponding pixel will have the same Y position, so the search can be limited along the x-axis and up to a maximum disparity.
The stereo matching methods that are considered in this thesis can be split into two categories, local based approaches and global based approaches, where local approaches mainly find the best fit disparity value by choosing the minimal aggregated cost of a window of pixels, while the global-based approaches into these steps [16]

- **Matching cost computation** SSD, SAD, NCC
- **Cost aggregation**
- **Disparity computation/optimization**
- **Disparity refinement (post processing)**

3.4.2 Matching Cost Computation

There are multiple ways to calculate the matching cost of a pixel pair [23]. The best pixel match will be assessed with a cost function, and the matching cost will be assessed by checking the Neighborhood pixels N for each pixel with a disparity value d, the matching cost will be computed, and then the most suited candidate will be used for the next stage.

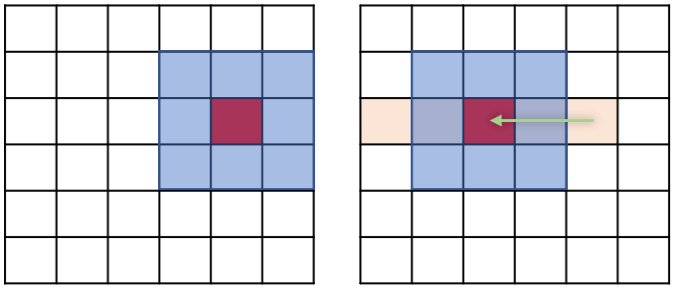


Figure 16: Matching cost computation of two pixels p (marked in red), with a position difference of disparity d (green arrow) in a local block matching algorithm assessed by checking the neighbourhood pixels N (marked in blue)

There are two methods used to determine the matching cost; first is a **parametric matching cost** which calculates cost based on the intensity values within the neighbourhood window, and the second is a **non-parametric matching cost** which uses only the local ordering of intensity

values within the neighborhood pixels, which is more robust against illumination changes.

3.4.3 Parametric Matching Cost Functions

Parametric matching cost functions provides a score for the pixel pair's dissimilarity, and it does that by directly comparing the pixel's intensity values, this provides the advantage of not needing to perform any computation before calculating the cost. However, it makes it particularly vulnerable to changes in illumination.
Illumination can be different between the image pair. Two common challenges that are common are the bias and the vignetting effect. The bias indicates that one image is brighter than the other 17, it usually occurs due to the sensors of the camera reacting differently to the illumination, while the vignetting effect occurs when the pixels at the centre of the image are brighter than the edges.

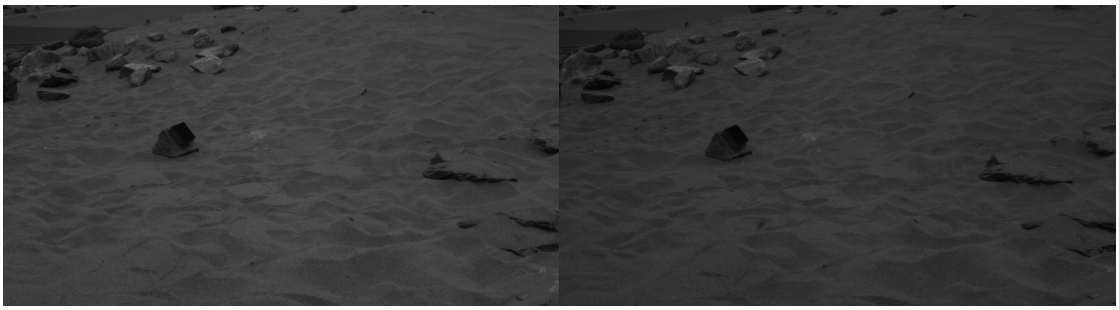


Figure 17: Illumination difference in the planetary I3DS dataset, showing a bias with the right image being darker than the left image

Different types of parametric matching cost functions have been proposed throughout literature [23] each with its advantages and disadvantages.

The Sum of Squared Difference (SSD)

The sum of squared difference is commonly used matching cost function; it is computationally efficient and provides a simple way to get the matching cost. However, it doesn't account for any of the illumination difference; it assumes brightness constancy thus doesn't account for bias and doesn't deal with the aperture problem and thus doesn't account for offset.

$$f_{SSD}(p,d) = \sum_{q \epsilon N(p)} (I_l(q) - I_r(q-d))^2 \tag{3.16}$$

Where p is the pixel coordinate, d is the disparity, I_l and I_r are the intensity values in the left and right image respectively, N is the neighbourhood bounded by the aggregation window used in the block matching, and q are the pixels within that window.

The Sum of Absolute Difference (SAD)

The sum of absolute difference is commonly used in the stereo matching algorithm and is solved without the need for any multiplication unlike the sum of squared difference, which makes it

well suited for embedded systems applications.

$$f_{SAD}(p,d) = \sum_{q \epsilon N(p)} \mid I_l(q) - I_r(q-d) \mid \tag{3.17}$$

However the same as the sum of squared difference it doesn't deal with the change in illumination, it doesn't account for illumination bias or offset.

The Zero-mean Sum of Absolute Difference (ZSAD)

The zero-mean sum of absolute difference calculate the mean intensity of the window and subtracts and subtracts it from every pixel in the neighbourhood.

$$f_{ZSAD}(p,d) = \sum_{q \epsilon N(p)} \mid (I_l(q) - \bar{I}_l(q)) - (I_r(q-d) - \bar{I}_r(q-d)) \mid \tag{3.18}$$

Where $\bar{I}$ is the mean intensity value of the neighborhood pixels.

$$\bar{I}(p) = \frac{1}{|N(p)|} \sum_{q \epsilon N(p)} I(q) \tag{3.19}$$

The zero-mean provides mitigation against brightness offset; however, it doesn't take into account the gain in intensity.

The Normalized Cross Correlation (NCC)

Normalized cross-correlation provides the ability to deal with the gain difference, and also optimal to dealing with gaussian noise; however, it doesn't deal with the brightness offset and blurs depth discontinuities.

$$f_{NCC}(p,d) = \frac{\sum_{q \epsilon N(p)} I_l(q) I_r(q-d)}{\sqrt{\sum_{q \epsilon N(p)} (I_l(q))^2 \sum_{q \epsilon N(p)} (I_r(q-d))^2}} \tag{3.20}$$

The Zero-mean Normalized Cross Correlation (ZNCC)

The Zero-mean Normalized Cross Correlation is the only function that accounts for both the brightness offset and the gain difference, which makes it the most accurate matching cost representation.

$$f_{ZNCC}(p,d) = \frac{\sum_{q \epsilon N(p)} (I_l(q) - \bar{I}_l(q))(I_r(q-d) - \bar{I}_r(q-d))}{\sqrt{\sum_{q \epsilon N(p)} (I_l(q) - \bar{I}_l(q))^2 \sum_{q \epsilon N(p)} (I_r(q-d) - \bar{I}_r(q-d))^2}} \tag{3.21}$$

However the zero-mean normalized cross correlation function is a very computationally demanding method, which makes it perform poorly only embedded systems and real-time applications.

3.4.4 Non-parametric Matching Cost Functions

The non-parametric matching cost functions calculates dissimilarity by using the ordering of intensities in the neighborhood pixels, this enables the function to efficiently deal with the illumination challenges mentioned earlier

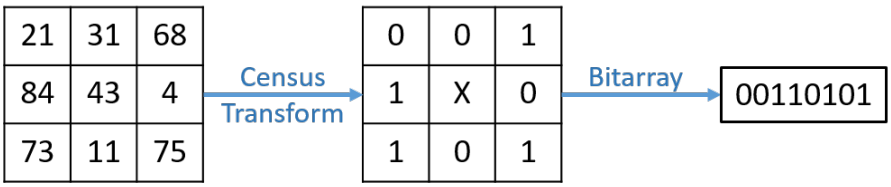


Figure 18: Given a pixel window census transform produces a bit array where every pixel represents if the neighbourhood pixel is smaller or larger

Rank

The rank transform [24] is a pixel transformation where each pixel is given a rank and the rank is calculated as the number of pixels smaller than this pixel.

$$I_{rank}(p) = \sum_{q \epsilon N(p)} T(I(q) < I(p)) \tag{3.22}$$

Where T is a function that returns 1 if the argument inside is true or 0 otherwise
The rank of the pixel can be used instead of the pixel intensity in the matching cost function, most commonly the sum of squared difference is used.

$$f_{Rank}(p,d) = f_{SSD}(I_{rank_l}(p), I_{rank_r}(p-d)) \tag{3.23}$$

Census

Similar to the rank function the census cost function doesn't operate on the intensity values, instead the order of intensities, which is represented in a bit string where every bit represents a pixel in the neighbourhood window, since it mainly deals with operations on bit arrays, this method is well suited for FPGA applications with its lookup tables.

$$I_{Census}(p,d) = BITSTRING_{N(p)}(T(I(p) < I(q))) \tag{3.24}$$

If the intensity of the neighborhood pixel is larger than the pixel in question, then the value is 1, otherwise it is 0. Applying the census transform result in a bit array with a size depending on neighbourhood window size. The resulting bit array from each image can then be used in a matching cost function the hamming distance,

$$f_{Census}(p,d) = \sum_{q \epsilon N(p)} HAMMING_n(I_{Census_l}(p), I_{Census_r}(p-d)) \tag{3.25}$$

The resulting value of the hamming distance given two bit arrays is the number of bits that are different from each other.

3.4.5 Cost aggregation

Cost aggregation is a process that is mainly used in local block matching, and it works by including the costs of multiple matches into a single cost.

3.4.6 Disparity Selection/Optimization

Local Approach

The local approach uses a method called Winner-takes-all (WTA), computing the disparity by choosing the disparity value that would offer the lowest cost.

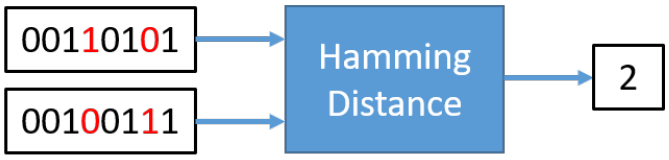


Figure 19: Given two bit arrays produced from a stereo image pair their matching cost is calculated using hamming distance by using an XOR operator and summing the bits

Global Approach

The global approach treats the disparity selection problem as a global optimization problem by using an energy function with a smoothness assumption. The changes in neighbouring disparities are penalized, therefore enforcing higher smoothness.

Semi-Global Approach

Semi-global matching first suggested by Hirschmuller [25] attempts to give similar performance as a global matching approach using only the information from a local window

4 Methods and Tools

This chapter presents the tools used for this thesis first all the boards used which feature both FPGA and GPU accelerator-based designs, and the second is the software tools used for the project including the software development kits by Xilinx and NVIDIA.

4.1 Presentation of the hardware

Two heterogeneous architectures were employed in the development of the stereo system and later compared.

- **The Zynq Ultrascale+** architecture 10, of which two models are used in this project, the **XCZU3EG** model used in the the Ultrazed-EG board, and the much larger **XCZU9EG** model 1 used in the TE0808 board, enabling the acceleration of a larger design within the programmable logic.
- **The Tegra X2** architecture 11, which was used in **The NVIDIA Jetson TX2** board

Three boards were used in this project and they are presented in this section.
Note: Each board's features presented in this section is taken from it's respective documentation.

4.1.1 Ultrazed-EG Starter kit

The UltraZed-EG Starter Kit 21 consists of the Ultrazed-EG IO Carrier Card with the UltraZed-EG System on module included and mounted on the board.

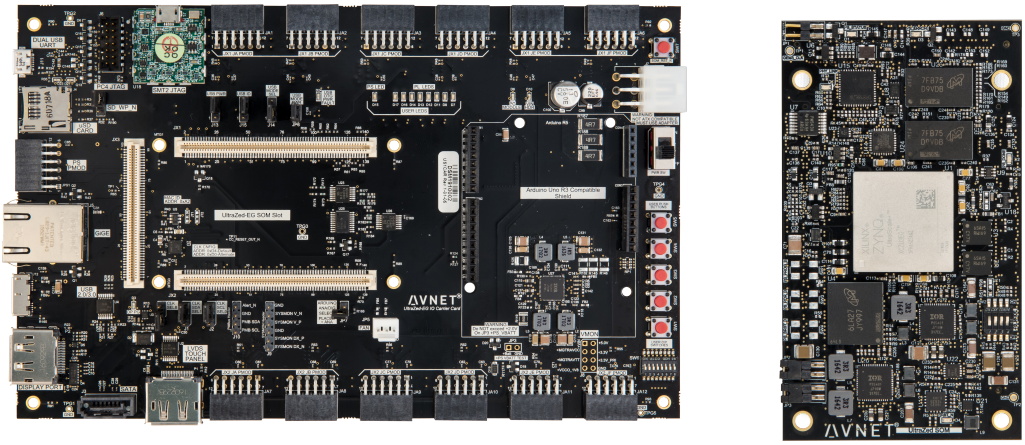


Figure 20: The Ultrazed-EG IO carrier card on the left and the Ultrazed-EG system on module on the right featuring an XCZU3EG chip [7]

System on module features

- Xilinx Zynq UltraScale+ XCZU3EG-SFVA625

- 2GB DDR4 SDRAM (in x32 configuration) with speed up to 2,133 Mbps
- Dual QSPI Flash (64MB)
- eMMC Flash (8GB, in x8 configuration)

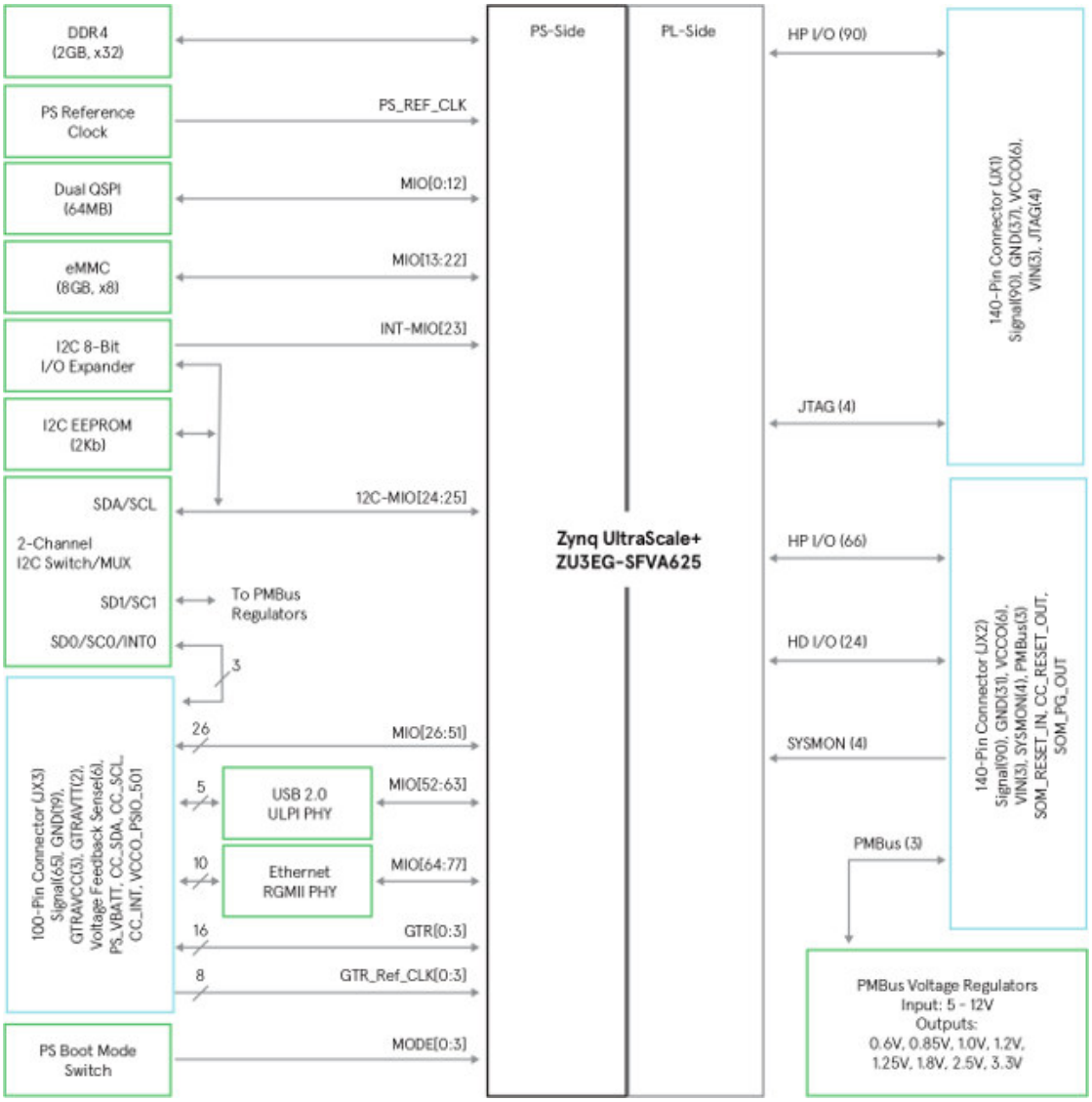


Figure 21: The Ultrazed-EG SoM block diagram[7]

Carrier board I/O interfaces

- 2 Display ports
- Ethernet Port
- MicroSD card slot
- 9 LEDs, 5 push buttons, 5 switches

- USB 2.0/3.0
- JTAG USB module

4.1.2 TE0808-04-09-2IE-S Starter Kit

The Trenz Electronic Starter Kit TE0808-04-09-2IE-S consists of a TE0808-04-09EG-2IE system on module featuring a **ZU9EG** Zynq Ultrascale+ chip mounted on a TEBF0808-04 base board.

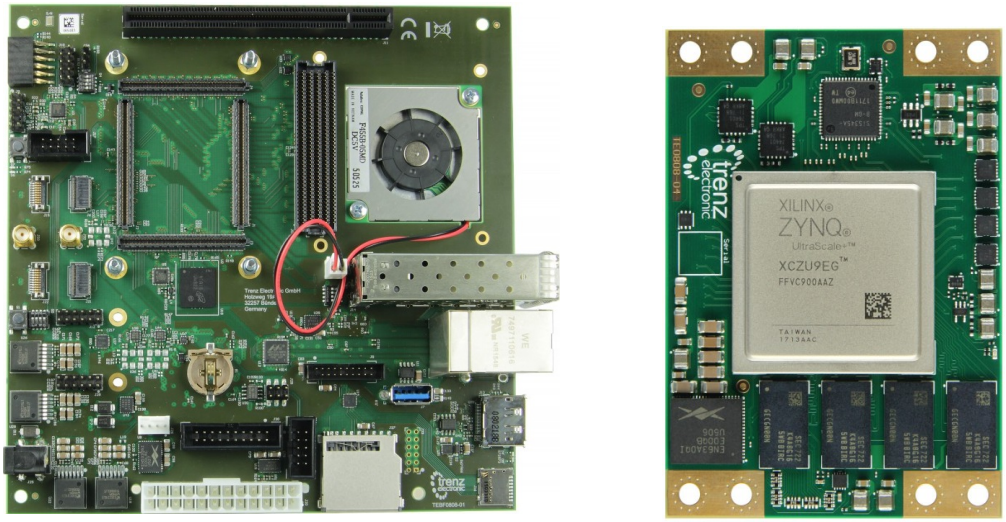


Figure 22: TE0808-04 carrier board on the left and the TE0808-04-09EG-2IE system on module on the right featuring an XCZU9EG chip [8]

The Module: TE0808-04-09-2IE The Trenz Electronic TE0808-04-09EG-2IE is a MPSoC module with industrial temperature grade integrating a Xilinx Zynq UltraScale+, 4 GByte DDR4 SDRAM with 64-Bit width, 128 MByte Flash memory for configuration and operation.

System on module key features

- Xilinx Zynq UltraScale+ XCZU9EG-2FFVC900I
- 4 GByte 64-Bit DDR4 SDRAM
- 128 MByte SPI Boot Flash (dual parallel)
- All power supplies on board, single 3.3V Power required
  - 14 on-board DC/DC regulators and 13 LDOs
  - LP, FP, PL separately controlled power domains
- Support for all boot modes (except NAND)

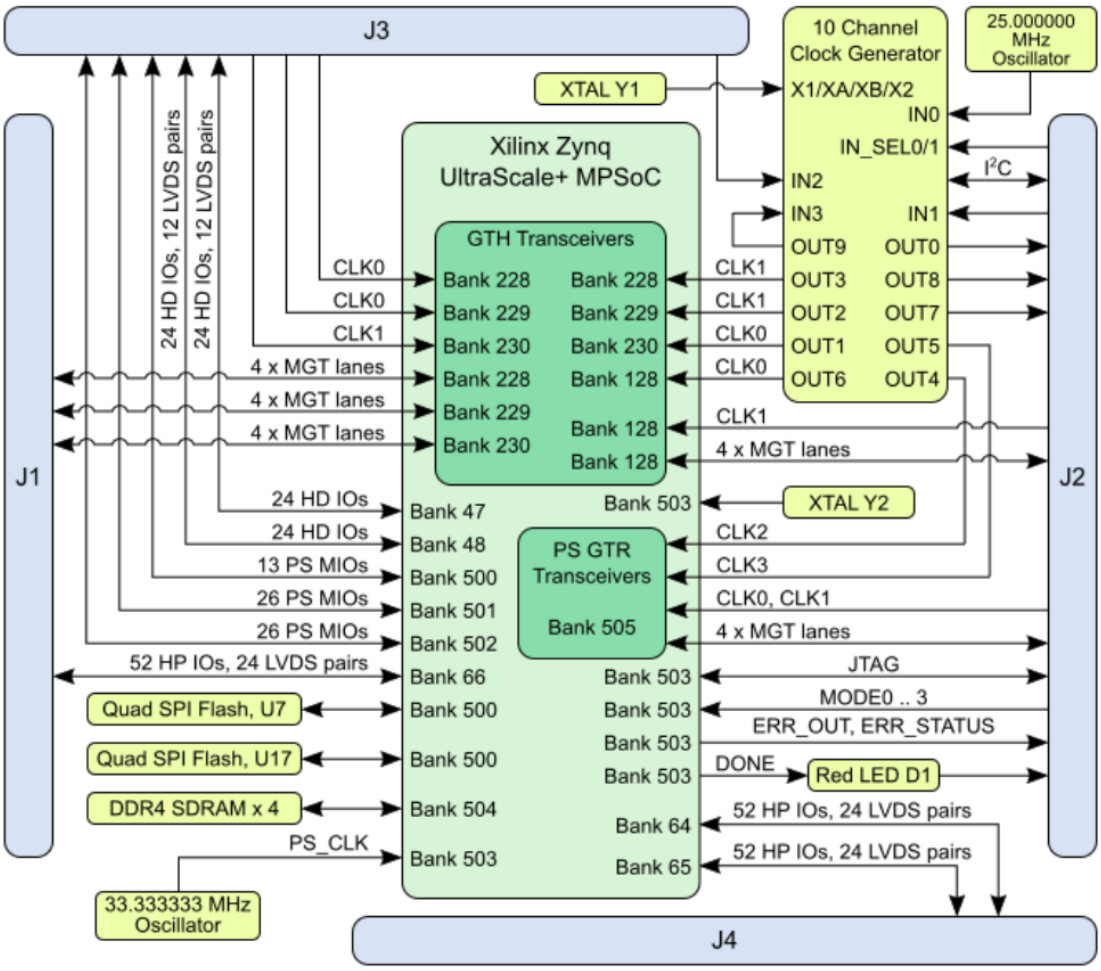


Figure 23: The TE0808-04-09 block diagram [8]

Carrier board key features

- Displayport
- 4x USB3.0/2.0 ports
- Gigabit ethernet port
- Quad programmable clock generator
- MicroSD socket
- 4 GB eMMC flash
- 2x JTAG/UART headers used for programming MPSoC

Both the **ZU3EG** and the **ZU9EG** chip share the same processing system, however the programmable logic is significantly larger enabling more functions to be moved to hardware.

	ZU3EG	ZU9EG
System Logic Cells (K)	154	600
Flip-Flops (K)	141	548
DSP Slices	360	2520
Block RAM (K)	7.6	32.1
Look Up Tables (K)	71	274

Table 1: Comparison of the ZU3EG model to the ZU9EG

4.1.3 Jetson TX2 developer kit

The NVIDIA Jetson TX2 Developer Kit is used in this project which features a **Jetson TX2 Module** which includes an NVIDIA Pascal architecture GPU and an ARM Cortex A57 quad-core CPU. It can run Ubuntu desktop, and it is used in this project to run the stereo system on both the CPU and GPU architectures.

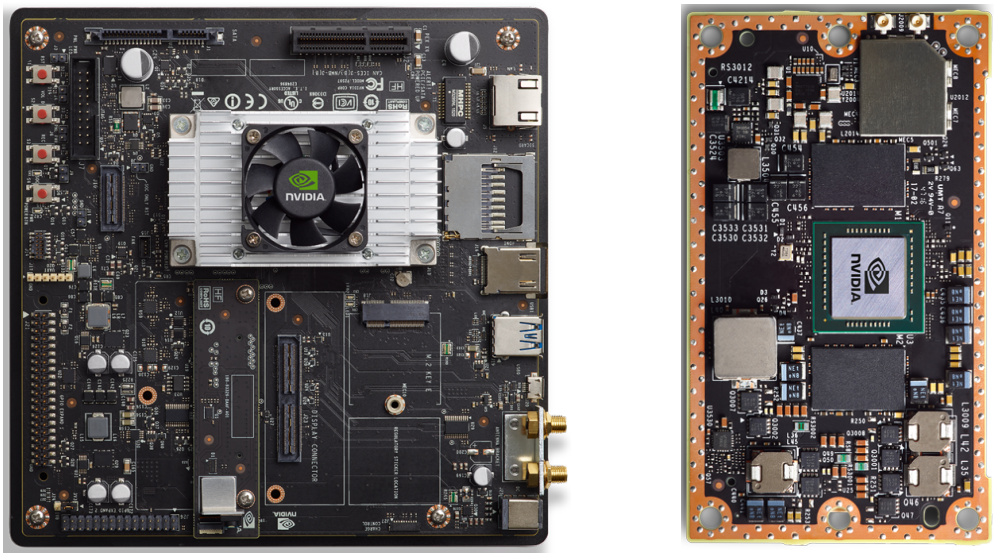


Figure 24: The Jetson TX2 development kit on the left and the Jetson TX2 system on module on the right [9]

Key features

- GPU NVIDIA Pascal™, 256 NVIDIA CUDA® cores
- CPU HMP Dual Denver 2/2MB L2 + Quad ARM® A57/2MB L2
- Memory 8 GB 128-bit LPDDR4 58.3 GB/s
- Video decode 4K x 2K 60 Hz Decode (12-bit support)
- Video encode 4K x 2K 60 Hz Encode (HEVC)
- CSI Up to 6 cameras (2 lane) CSI2 D-PHY 1.1 (2.5 Gbps/lane)
- Display HDMI 2.0 / eDP 1.4 / 2x DSI / 2x DP 1.2
- Connectivity 1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
- Networking 1 Gigabit Ethernet

- PCI-E Gen 2 | 1x4 + 1x1 OR 2x1 + 1x2
- Data Storage 32 GB eMMC, SDIO, SATA
- Other CAN, UART, SPI, I2C, I2S, GPIOs
- USB USB 3.0 + USB 2.0
- Power 7.5 W / 15 W

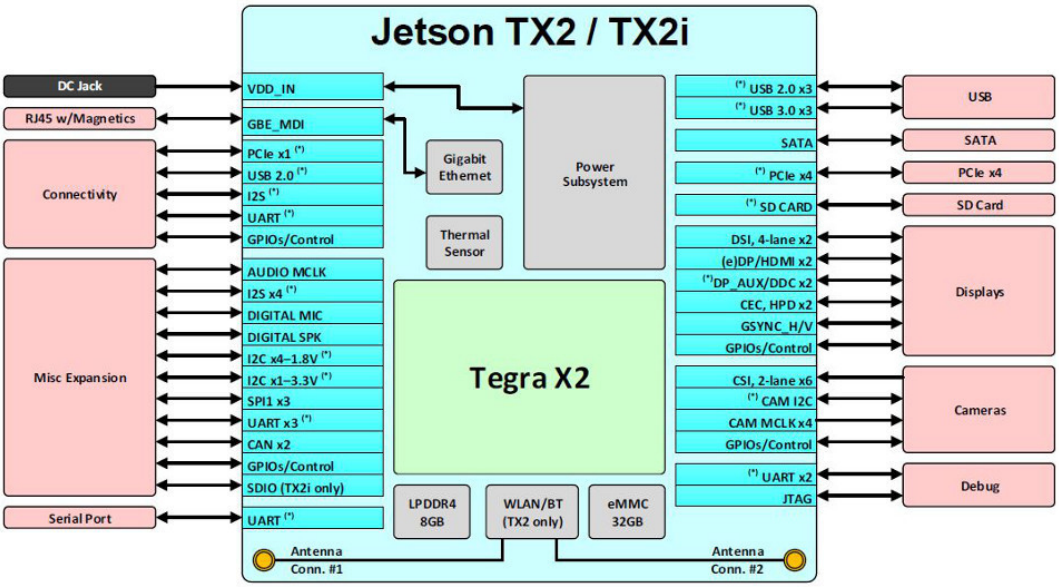


Figure 25: The Jetson TX2 block diagram [9]

4.2 Software tools

4.2.1 Vivado Design Suite

The Vivado Design Tools is a software suite by Xilinx used for synthesis and analysis of hardware description language designs; it's used in this project for the synthesis of the hardware platform A used for the development of the FPGA accelerated stereo system.

4.2.2 PetaLinux tools

The PetaLinux tools is a toolchain by Xilinx based on the Yocto project, and it is used to build a customised embedded Linux operating system for Xilinx chips, including the Zynq Ultrascale+ board used in this project.

4.2.3 SDx/SDSoC

SDSoC is a development environment by Xilinx recently combined with SDAccel to make the SDx integrated development environment. The IDE enables hardware, software co-design with embedded C/C++/OpenCL application development for Zynq SoC and MPSoC devices. The function can be cross-compiled to run on the ARM processor, or it can be marked for acceleration, and synthesised to run on the FPGA using Xilinx's high-level synthesis tool.

4.3 Libraries

4.3.1 OpenCV

Open source computer vision library [26] (OpenCV), aimed for the application of real-time computer vision, it is considered as the de-facto standard API in computer vision, it is free to use and cross-platform, which is used in this project for the embedded stereo system running on an ARM processor.

4.3.2 CUDA

CUDA [20] is a parallel computation API, developed by NVIDIA, which enables a higher degree of abstraction for building parallel computing applications on CUDA-enabled hardware, such as the NVIDIA Jetson TX2 used in this project. CUDA is used in conjunction with OpenCV to build the GPU-accelerated stereo system.

4.3.3 libsgm

libSGM [27] is a Semi-Global block Matching CUDA implementation library developed by fixstars. The OpenCV::CUDA libraries lacked a semi-global block matching GPU-accelerated function, so the Libsgm libraries were chosen as it is integrated with OpenCV and CUDA.

4.3.4 XFopenCV

Xilinx Fast OpenCV [28] is a library by Xilinx, developed for use in the SDx/SDSoC IDE, and it offers hardware acceleration ready computer vision libraries.

4.3.5 Benchmark

Benchmark[29] is a library by Google, that can be used to benchmark code, and it is used to benchmark the computational time of the stereo systems.

5 Stereo System Calibration

To be able to process stereo image pairs and generate depth information. The calibration stage is needed to extract the intrinsic and extrinsic parameters, which can then be used to rectify the input stereo pairs.

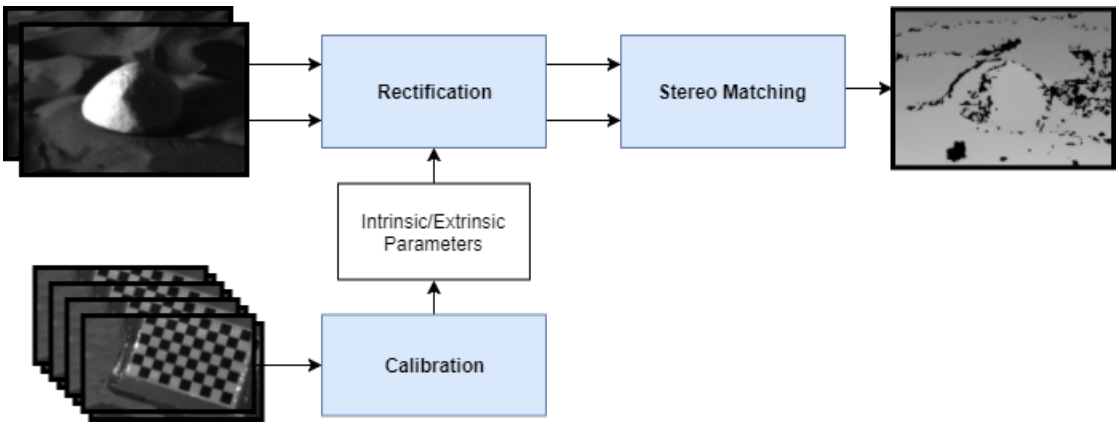


Figure 26: High-level view of the overall stereo system; consisting of calibration, rectification and stereo matching blocks

5.1 Calibration

In order to generate a fully rectified image the stereo system's intrinsic, extrinsic and distortion parameters need to be known, these parameters are calculated in the stereo system calibration phase. The system was first calibrated using the Matlab stereo camera calibrator, however, due to Matlab using a different frame of reference than OpenCV, the output undistortion and rectification parameters didn't provide correct results.

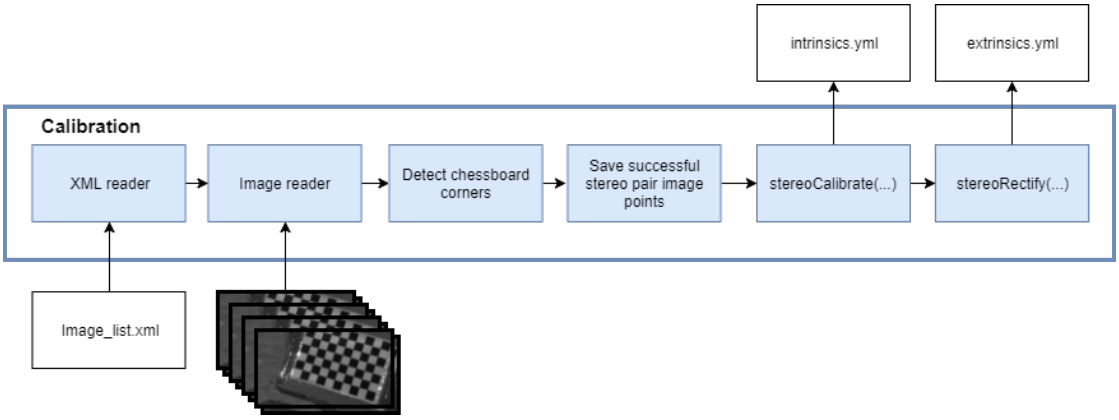


Figure 27: The block design of the stereo calibrator C++ app

The system was then recalibrated using OpenCV in C++, not only to calculate the rectification/undistortion parameters into a usable form without requiring further operations but also to store them in YML files, which are then read later in the stereo matching algorithm whether running on a CPU or a Hardware accelerated embedded system. The stereo system calibration is based on the technique by Zhang et al. [30]. where calibration paramaters are extracted from multiple images of a checkerboard pattern from different viewpoints and then the checkerboard corners are extracted. From the known checkerboard shape and size, the undistorted positions of these corners can deduced.
The stereo camera used for the I3DS planetary dataset was calibrated using 180 stereo image pairs 31. The checkerboard was in put in different positions as shown in the figure 29

5.1.1 Camera Parameters

Left Camera

First the **stereoCalibrate(...)** function computes the distortion coefficients, which is explained in the equation 3.10, it's used in the rectification process to remove the distortions caused by the lens.

$$k_1 = -0.362060 \quad k_2 = 0.723516 \quad k_3 = 0 \quad k_4 = 0 \quad k_5 = 0 \quad k_6 = 0.779572$$

Further more the focal length is denoted as f_x and f_y and the principle point's location c_x and c_y relative to the reference frame.

$$f = \begin{bmatrix} f_x & f_y \end{bmatrix} = \begin{bmatrix} 1862.76 & 1927.20 \end{bmatrix} \qquad c = \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} 956.970 \\ 535.796 \end{bmatrix}$$

the camera matrix in the "old coordinate system" before the stereo rectification takes effect, is denoted as **K**.

$$\mathbf{K} = \begin{bmatrix} 1862.76 & 0 & 956.970 \\ 0 & 1927.20 & 535.796 \\ 0 & 0 & 1 \end{bmatrix}$$

Next the **stereoRectify(...)** function computes the necessary transformations to make the image planes lie on the same plane, and to make all the epipolar lines parallel to significantly reduce

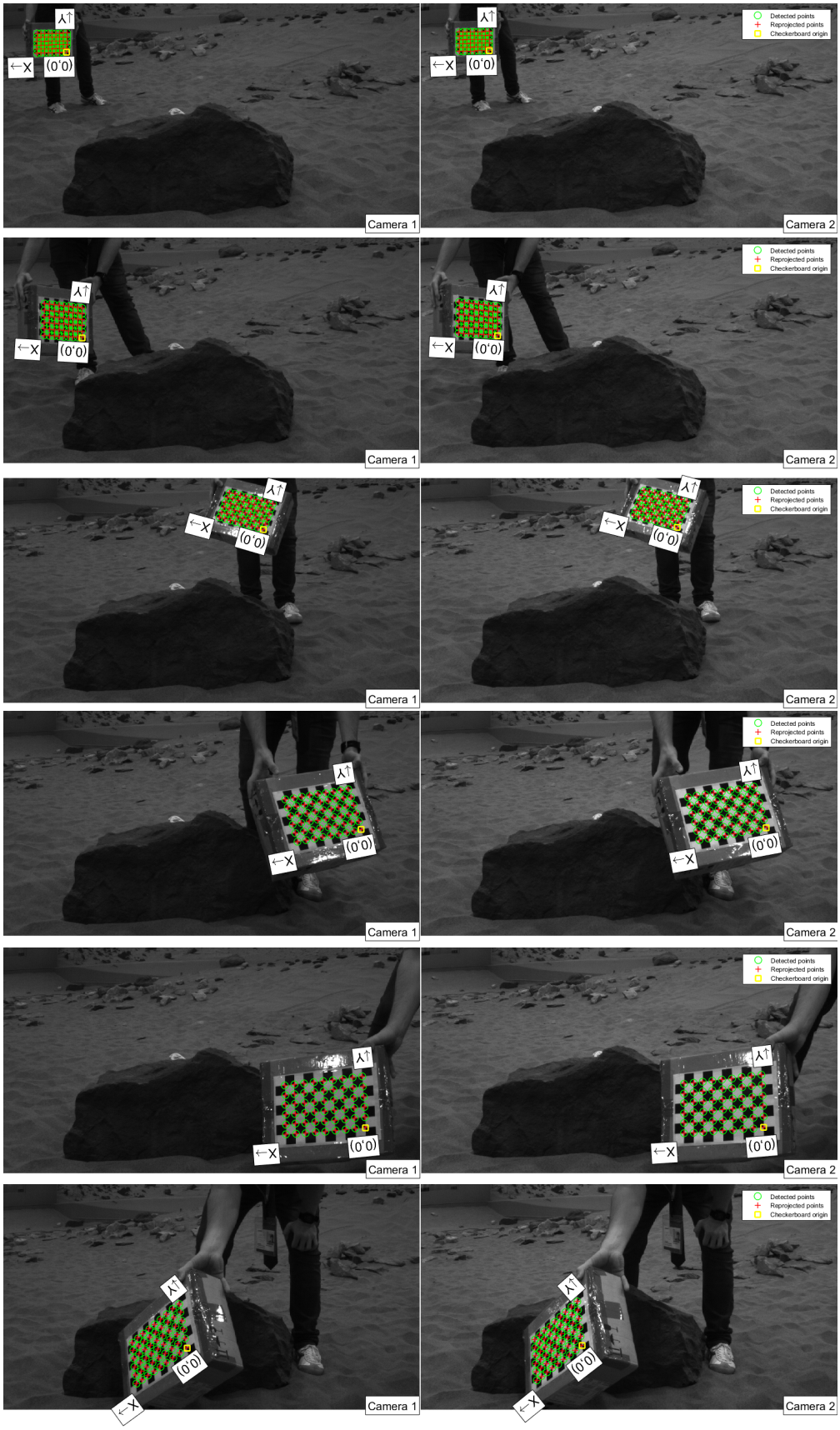


Figure 28: Sample set of stereo image calibration pairs taken during I3DS planetary use-case validation in the Airbus artificial Mars landscape in Stevenage.

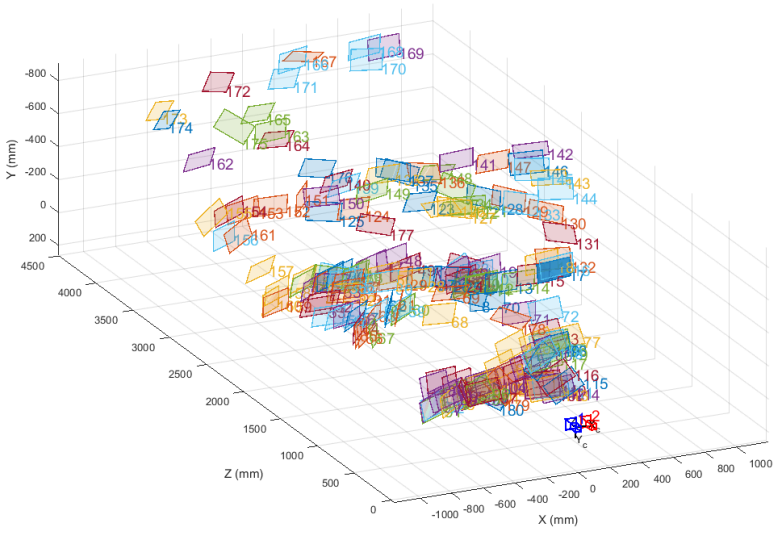


Figure 29: Figure generated from matlab showing the relative position and orientation of every checkerboard snap to the cameras

the search area in the stereo matching function. The function takes in the undistortion matrices from stereoCalibrate(...) as input. and then produces two rotation matrices $\mathbf{R_l}$ and $\mathbf{R_r}$

$$\mathbf{R_l} = \begin{bmatrix} 0.999721 & 0.002487 & -0.023487 \\ -0.0025143 & 0.999996 & -0.001099 \\ 0.023484 & 0.001158 & 0.999723 \end{bmatrix}$$

The function also generates two projection matrices in the new coordinate system $\mathbf{P_l}$ and $\mathbf{P_r}$.

$$\mathbf{P_l} = \begin{bmatrix} 0.001615 & 0 & 0.001061 & 0 \\ 0 & 0.001615 & 0.053136 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Right Camera

The undistortion and rectification parameters are also calculated for the right image and documented here.

The homogenous transformation matrix $\mathbf{T_{RL}}$ describing the position and orientation of the right camera with reference to the left camera.

$$\mathbf{T_{RL}} = \left[\mathbf{R}_{3x3}|\mathbf{T}_{3x1}\right] = \begin{bmatrix} 0.99998 & -0.00078 & 0.00557 & -0.09652 \\ 0.00079 & 0.999997 & -0.00218 & -0.00032 \\ -0.00556 & 0.00218 & 0.99998 & 0.00280 \end{bmatrix}$$

The distortion coefficients denoted as k_n and the

$$k_1 = -0.362060 \quad k_2 = 0.693283 \quad k_3 = 0 \quad k_4 = 0 \quad k_5 = 0 \quad k_6 = 0.835517$$

$$f = \begin{bmatrix} f_x & f_y \end{bmatrix} = \begin{bmatrix} 1862.76 & 1927.20 \end{bmatrix} \qquad c = \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} 1034.00 \\ 529.964 \end{bmatrix}$$

$$\mathbf{K_r} = \begin{bmatrix} 1862.76 & 0 & 1034.00 \\ 0 & 1927.20 & 529.964 \\ 0 & 0 & 1 \end{bmatrix}$$

The rectification transform (rotation matrix) for the right image.

$$\mathbf{R_r} = \begin{bmatrix} 0.999572 & 0.003337 & -0.029050 \\ -0.003307 & 0.999993 & -0.001101 \\ 0.029053 & 0.001004 & 0.999577 \end{bmatrix}$$

The projection matrix in the new (rectified) coordinate systems for the right image.

$$\mathbf{P_r} = \begin{bmatrix} 0.001615 & 0 & 0.001061 & -0.015600 \\ 0 & 0.001615 & 0.053136 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The perspective transformation matrix which maps the disparity to depth; which can be used to convert a disparity map to a 3D surface.

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & -0.001061 \\ 0 & 1 & 0 & -0.053136 \\ 0 & 0 & 0 & 0.001615 \\ 0 & 0 & 0.103560 & 0 \end{bmatrix}$$

5.2 Rectification

Unlike the calibration process, the rectification process will need to be implemented in every architecture to rectify the input image stream in real-time, as the stereo matching algorithms assume rectified images.

5.2.1 cv::InitUndistortRectify(...)

This function takes as input the parameters generated in the calibration stage, which are saved in the intrinsics and extrinsics yml files, and it is used to compute the joint transformation for undistortion and rectification using the rectification processes from 3.7 until 3.14.
The function then generates a map which can be used in the remap(...) function to convert the live stereo pair feed, into a rectified image. The generated map computes each pixel's new position in the destination image which is both rectified and undistorted.

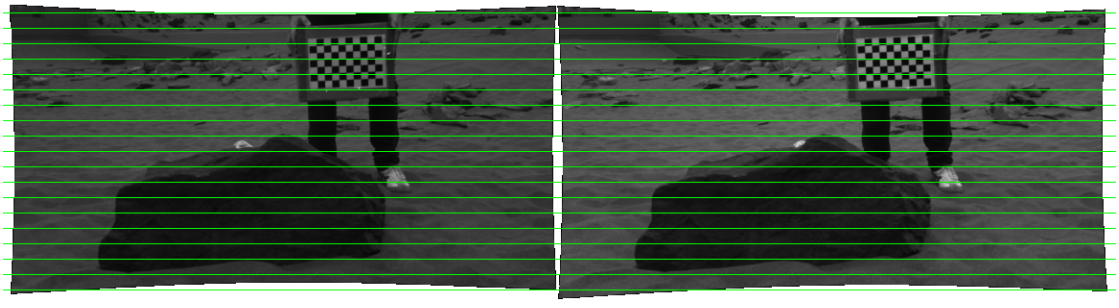


Figure 30: Rectification of the stereo pair from the I3DS dataset

The generated map applies a new camera view. which is rotated according to R_l and R_r to align the epipolar lines horizontally, this function is called twice for the right and left images.

5.2.2 cv::Remap(...)

The remap function is called twice for the left and right images, applying a generic geometrical transformation to an image, taking as input the map matrices generated by the InitUndistortRectify function, and the stereo image pair, remapping the stereo images into a corresponding undistorted and rectified image.

$$\mathbf{dst}(x, y) = \mathbf{src}(map_x(x, y), map_y(x, y))$$

Pixel values with non-integer coordinates are computed using the interpolation method explained in 3.15.

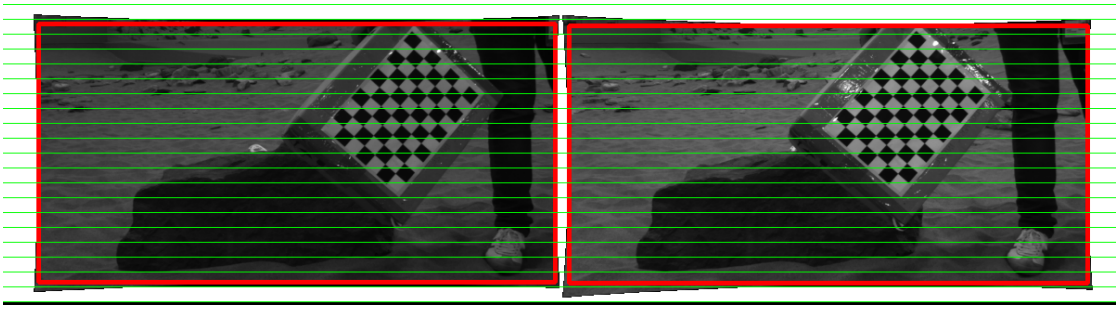


Figure 31: Rectification and taking a region of interest of a stereo pair from the I3DS for correct disparity map generation

After the image is rectified and undistorted, a region of interest is selected to crop the image into the largest possible rectangle in both images.

6 Embedded Stereo System Implementation

The source code developed for this project is available on Github using this link.

6.1 Specification

The aim of this chapter is the development of an embedded stereo system on different architectures, first the FPGA accelerated stereo system demonstrated, as it highlights the optimization of the stereo system, and then the CPU and GPU implementations are shown afterwards

6.2 FPGA development environment

XFopenCV are libraries developed by Xilinx, it takes advantage of the high level synthesis tools, to let the developer work with hardware accelerated image processing functions on a relatively high abstraction.

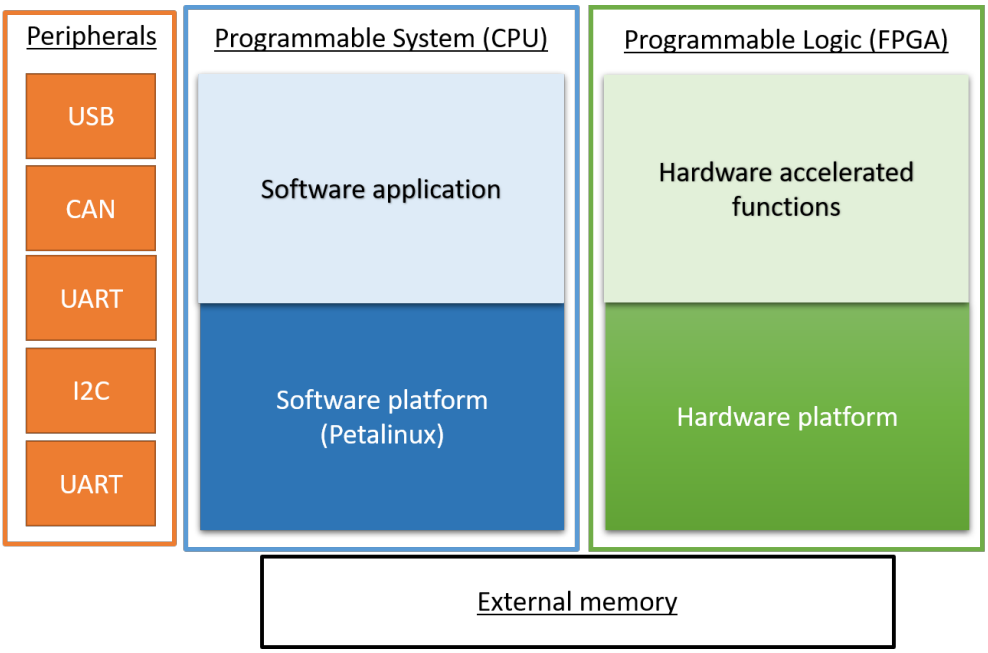


Figure 32: High-level view of the software system implemented on the Ultrascale+ architecture

Xilinx's SDx/SDSoC IDE enables hardware/software co-design, to be able to develop the hardware accelerated application, an SDSoC platform is needed. It consists of a hardware platform and a software platform, for a custom board like the one used in this project, developing a cus-

tom platform might be needed. The TE0808 has an SDSoC platform available in the website's resources, but it lacks an operating system.
The process of generating an SDSoC platform can be long, but the steps done to get the board ready for HW/SW stereo system development is documented in the appendix; the hardware platform A and the software platform B.

6.3 FPGA accelerated functions

6.3.1 xf::InitUndistortRectifyMapInverse

This function generates the remap matrices that rectify the image in real time, however, due to it being build for use in the FPGA this function has one difference with its openCV equivalent is that the inverse of the rotation matrix needs to be computed on the CPU, before being sent to the hardware.

6.3.2 xf::remap

The remap matrix takes the real-time image feed, and the remap matrix, to generate the rectified image.

6.3.3 xf::stereoBM

The stereoBM implementation in XFopenCV consists of preprocessing and disparity estimation, the preprocessing applies a sobel filter, and the matching cost is estimated using the sum of absolute difference 3.17, and the disparity is obtained using winner takes all, there is also a minimum uniqueness for the disparity to be accepted otherwise it will be set to zero.

6.3.4 xf::SemiGlobalBM

The semiGlobalBM implementation in xfOpenCV, consists of census transform 18 with the Hamming distance 19 to calculate the matching cost. with the optimization block being based off the Hirschmuller [25] approach.

6.4 FPGA optimizations

The FPGA has the unique advantage for its high degree of parallelism, and allow for the design of pipelined computational units, in this section

6.4.1 Stereo matching parallel units

The XFopenCV stereo matching functions allows control over its degree of parallelism, by controlling the number of parallel units that compute and aggregate the matching for a number of disparities in parallel, improving the performance the frame rates of the stereo system, however increasing the number of parallel units increase the resource usage, thus increasing the power consumption.

Resource	Used	Total	% Utilization
DSP	99	2520	3.93
BRAM	228	912	25
LUT	101268	274080	36.95
FF	90538	548160	16.52

Resource	Used	Total	% Utilization
DSP	99	2520	3.93
BRAM	228	912	25
LUT	138711	274080	50.61
FF	109187	548160	19.92

Resource	Used	Total	% Utilization
DSP	99	2520	3.93
BRAM	228	912	25
LUT	242361	274080	88.43
FF	141692	548160	25.85

Figure 33: Resource usage of a stereo system using 16, 32 and 80 parallel computational units respectively

Since every parallel unit will be computing a range of diparities, the number of disparities has to be divisible by the number of parallel units, XFopenCV's default is 16, the design was synthesised multiple time to find that the fpga chip can handle up to 80 parallel units, thus significantly improving the processing time.

6.4.2 The default stereo system

The default design for the stereo system is shown in the following code snippet.

```
    void stereoHW_stereoSystem(...){
    #pragma HLS INTERFACE m_axi depth = 9 port = cameraParmHW offset = direct
        ↪ bundle = cameraParam

    xf::InitUndistortRectifyMapInverse<...>(cameraParmHW, mapxL, mapyL);
    xf::remap<...>(left, leftRemapped, mapxL, mapyL);

    xf::InitUndistortRectifyMapInverse<...>(cameraParmHW, mapxR, mapyR);
    xf::remap<...>(right, rightRemapped, mapxR, mapyR);

    xf::StereoBM<...>(right, rightRemapped, dispMat, state);
}
```

The default stereo system implementation 34 during real-time operations, starts by moving the stereo pair images from the CPU to the FPGA by moving the images from cv::Mat to xf::Mat. Then InitUndistortRectifyMapInverse block is called twice consecutively for each image, and then the same for the remap block is called twice and then finally the disparity map is generated in the stereo matching block.

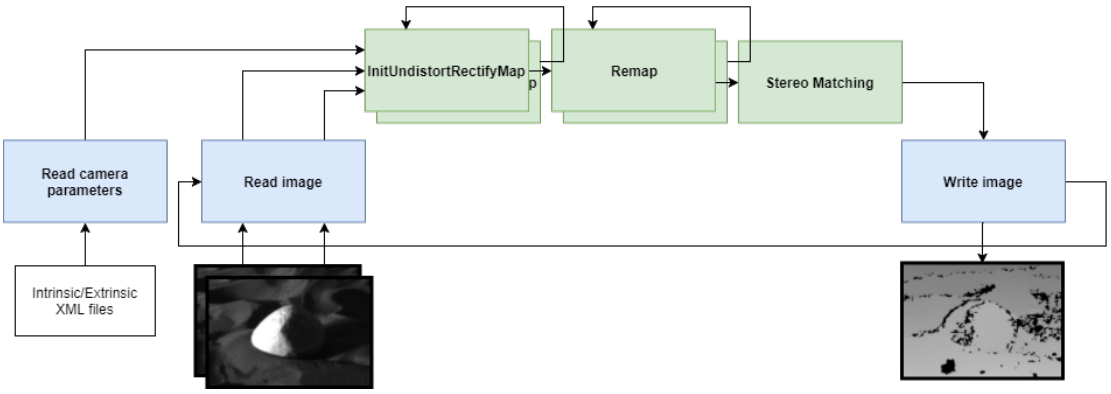


Figure 34: The default stereo system implementation, with the functions implemented on the CPU side marked in blue and the functions implemented on the FPGA side marked in green

6.4.3 Remap matrices storage

Since the InitUndistortRectifyMapInverse doesn't operate on the real-time image feed, it isn't required to be called for every stereo pair instead, the mapxL, mapyL, mapxR, mapyR can be stored with xf::Mat in the programmable fabric.

```
void stereoHW_ComputeRemap(...){
#pragma HLS INTERFACE m_axi depth = 9 port = cameraParmHW offset = direct
    ↪ bundle = cameraParam

    xf::InitUndistortRectifyMapInverse<...>(cameraParmHW, mapxL, mapyL);
    xf::InitUndistortRectifyMapInverse<...>(cameraParmHW, mapxR, mapyR);
}
```

```
void stereoHW_RectifyLocalBlockMatch(...){
    xf::remap<...>(left, leftRemapped, mapxL, mapyL);
    xf::remap<...>(right, rightRemapped, mapxR, mapyR);

    xf::StereoBM<...>(right, rightRemapped, dispMat, state);
}
```

and then the InitUndistortRectifyMapInverse block can be called only once 35 in the initialliza-tion of the stereo system.

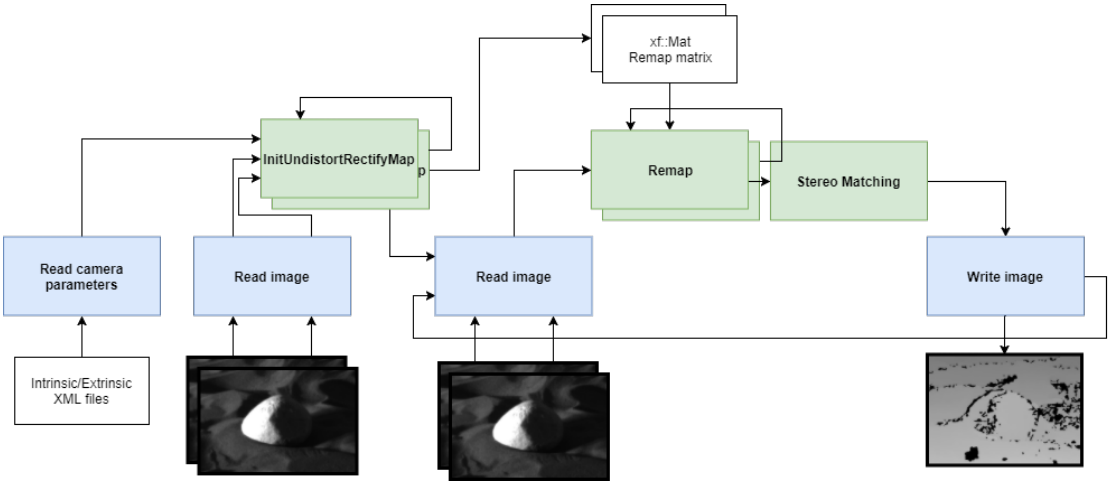


Figure 35: Block design of the stereo system with the InitUndistortRectifyMapInverse function moved to the initialization phase

6.4.4 Parallel remapping

Taking full advantage of the FPGA's inherently parallel architecture, two remap blocks are generated to rectify and undistort both images in parallel, and it can be achieved using the "SDS resource" pragma.

```
#pragma SDS resource(<ID>)
```

Once the SDS resource pragma is declared before a function, it signals that this hardware function will be synthesized more than once, and when the same function is called a second time with a different ID, a second instance of the hardware block will be synthesized.

```
    void stereoHW_RectifyLocalBlockMatch(...)
{
#pragma SDS resource(1)
    xf::remap<...>(left, leftRemapped, mapxL, mapyL);

#pragma SDS resource(2)
    xf::remap<...>(right, rightRemapped, mapxR, mapyR);

    xf::StereoBM<...>(right, rightRemapped, dispMat, state);
}
```

The stereo system now has two instances of the remap hardware functions 36 so every time a stereo pair is fed into the FPGA, both images automatically get rectified and undistorted before matching.

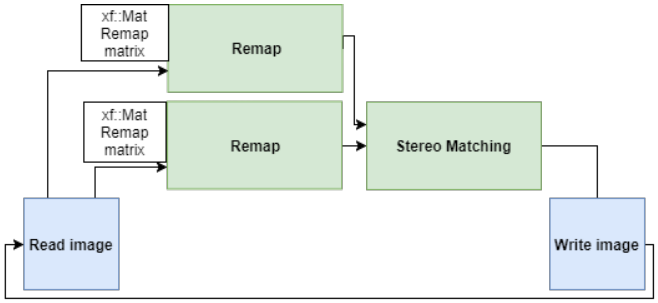


Figure 36: Block design of the stereo system with parallel remap, with the functions implemented on the CPU side marked in blue and the functions implemented on the FPGA side marked in green

6.4.5 Pipelining

Even with parallelized remap hardware functions, it takes 18 ms to compute on 100 MHz clock frequency, and 12 ms to compute using 150 MHz clock frequency. It is possible to remove the impact on the frame rates by pipelining the process. The functions can then be pipelined by calling them asynchronously, and this can be achieved using the SDS async and wait pragmas

```
#pragma SDS async(<ID>)
```

```
#pragma SDS wait(<ID>)
```

The pragma SDS async launches a hardware function asynchronously to run its computation in the background, and then immediately moves to the next function, and then moves forward with execution, up until it reaches an SDS wait with the same ID and waits.

```
void stereoHW_RectifyLocalBlockMatch(...){
#pragma SDS async(1)
#pragma SDS resource(1)
    xf::remap<...>(left, leftRemapped, mapxL, mapyL);

#pragma SDS async(2)
#pragma SDS resource(2)
    xf::remap<...>(right, rightRemapped, mapxR, mapyR);

#pragma SDS wait(1)
#pragma SDS wait(2)
#pragma SDS wait(3)
#pragma SDS async(3)
    xf::StereoBM<...>(right, rightRemapped, dispMat, state);
}
```

The code snippet shown is the final design of the real-time stereo system, and the same stereo system was also implemented to perform semi-global block matching, by replacing StereoBM with SemiGlobalBM.

```
    xf::SemiGlobalBM<...>(right, rightRemapped, dispMat, state);
```

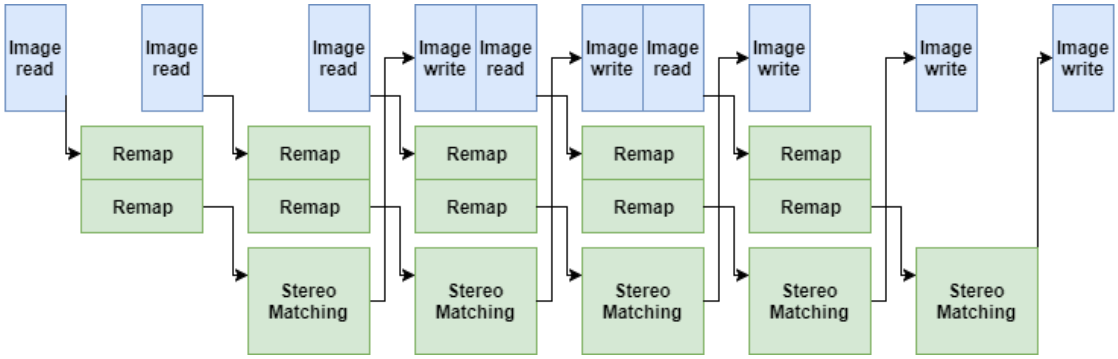


Figure 37: Final block design the pipelined stereo system, with the functions implemented on the CPU side marked in blue and the functions accelerated in hardware marked in green

The asynchronous launch of hardware functions allowed for the pipelining of the stereo system 37, and implemented a solution that is only bottlenecked by the computation of the stereo matching block.

6.5 GPU accelerated stereo system

The NVIDIA Tegra TX2 Jetson 25 Heterogeneous architecture allows the stereo system to be developed for an embedded CPU (2 GHz quad-core ARM Cortex A57) and an embedded GPU (1.2 GHz 256 cores NVIDIA Pascal) allowing both architecture to be tested.

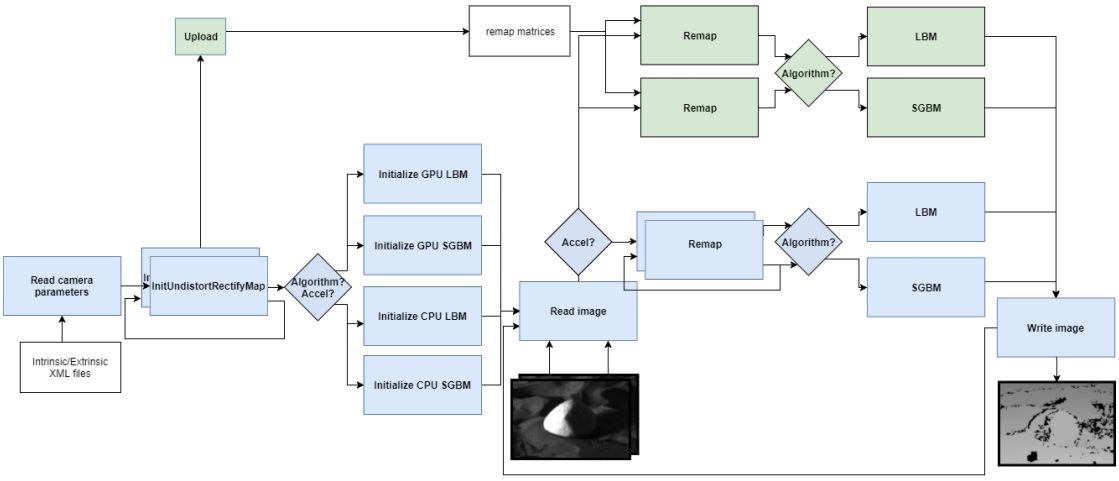


Figure 38: The implementation of the GPU accelerated stereo system, with the functions implemented on the CPU side marked in blue and the functions implemented on the GPU side, marked in green, accel? checks if the function is marked for acceleration in GPU, and Algorithm? checks if to run Local block matching or Semi-global block matching

6.5.1 The implementation

The stereo system was developed to be optimized with similar optimizations to the FPGA accelerated stereo system, the final design shown in 38.
The OpenCV Cuda implementation doesn't have a stereo semi-global block matching implementation, so the libsgm 4.3.3 was used. This GPU semi-global block matching approach was capped at 128 disparity, which does have an impact on the matching's algorithm ability to stereo match near objects with high disparity values, which will have an impact on this approach's accuracy in the results section.

6.6 GPU optimizations

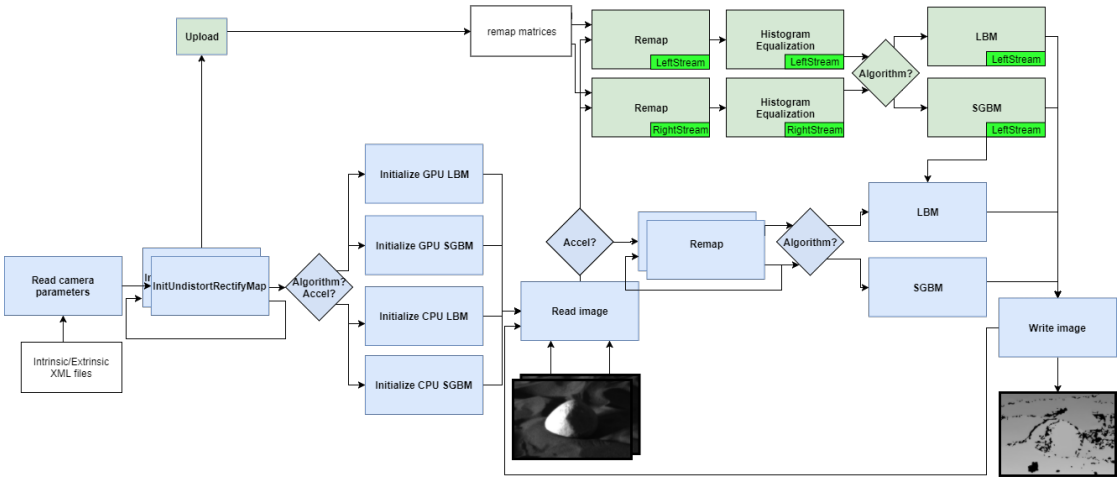


Figure 39: The implementation of the GPU accelerated stereo system with asynchronous execution and robustness towards illumination bias

To be able to take full advantage of the CPU performance, the following shell commands and shell script disable the dynamic frequency scaling, allowing both the CPU and GPU run on maximum performance during the performance tests.

```
nvpmodel -m 0
sudo jetson_clocks
```

The CUDA stereo matching implementation lacked robustness against illumination bias as explain in 17 in the stereo pairs. So a histogram equalization block was implemented on the GPU side for both images, due to pipelining it did not affect the frame rates; however, it did have a minimal effect on the latency.
To make sure the CPU doesn't wait for the GPU's computations to complete; asynchronous execution is achieved with **GPU::Stream**. First, only a single stream was used for all the functions, but that caused consecutive execution of the GPU accelerated function functions, so a second stream was added for the right image handling 39, allowing for pipelined execution 37.

7 Benchmarks and Results

Two stereo pipeline implementations are to be tested, the first is based on local block matching method, and the other is implementing semi-global block matching, both implementations are the most common implementations for real-time resource-constrained devices.
The stereo pipeline implementations are tested on three different architectures first the embedded CPU architecture, implemented on a 2 GHz quad-core ARM Cortex-A57 processor, second is the embedded GPU architecture, implemented on an NVIDIA Pascal GPU. Both the CPU and GPU are tested on an Nvidia Jetson TX2 board 25. The third architecture tested is the FPGA on a Zynq Ultrascale+ architecture XCZU9EG chip, on a Trenz TE0808-04-09EG-2IE 22 SoM and it's carrier board.
The embedded CPU implementation was done using the OpenCV libraries cross-compiled for ARM platform. The embedded GPU implementation was done using the OpenCV::Cuda libraries, however, the Cuda libraries lacked an official semi-global block matching, so the libsgm library for OpenCV::Cuda was used. The FPGA implementation was developed using the XFopenCV libraries by Xilinx.

7.1 Test parameters

For the benchmark tests implemented the stereo pipeline implementation will need to be able to process FHD 1920x1080 stereo pair images. The dataset images used in the tests are of either FHD or higher quality. However, images in the dataset that don't have the same aspect ratio are scaled so that at least either it's width or length have the same value of a 1920x1080 image.
The number of disparities in the stereo matching implementations is set to 160, which was obtained through trial and error over the I3DS and Middlebury datasets. 160 maximum disparity provided good quality disparity maps for 1920x1080 stereo pair images. however the current Cuda libsgbm implementation of semi-global block matching on the GPU has a maximum value of 128 for the number of disparities, so only for the GPU SGBM implementation it's set to its maximum value of 128.
The LBM and SGBM implementations on CPU, GPU and FPGA are tested and compared for their power consumption, stereo matching accuracy and processing time from which the maximum frames per second can be deduced.

7.2 The I3DS dataset

The I3DS dataset was taken in an artificial Mars landscape at Airbus in Stevenage to simulate stereo images from a planetary exploration rover. The images were then processed using all six implementations explained earlier for their real-time performance.

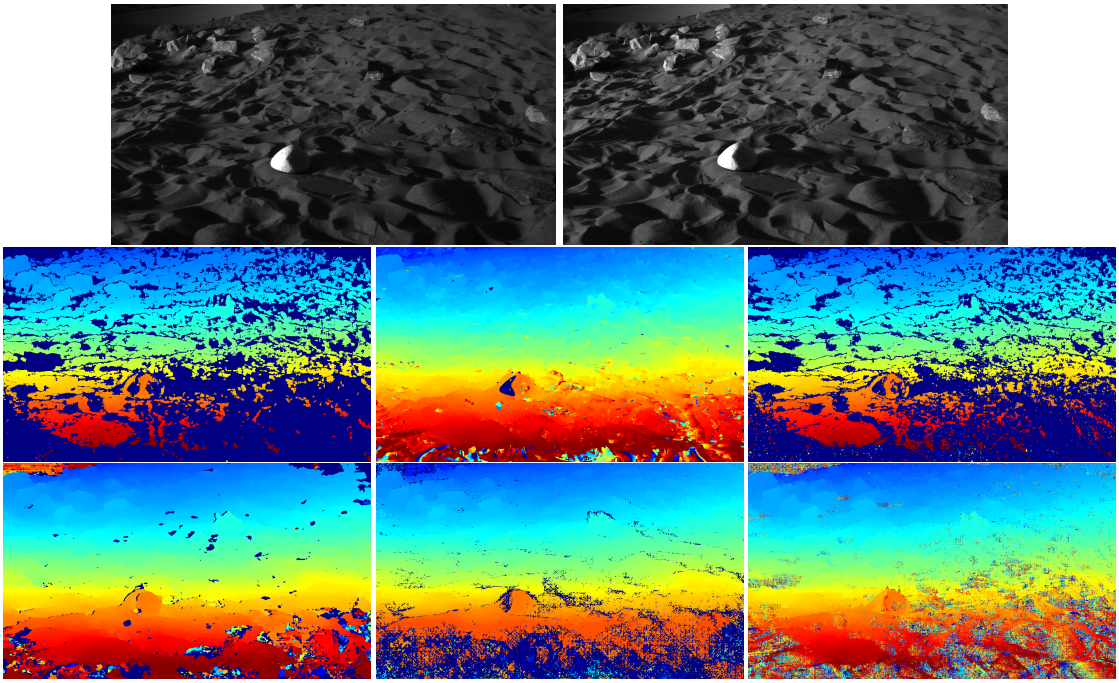


Figure 40: The first row shows a stereo pair left and right images respectively, the second row shows the disparity maps from local block matching implementations on CPU, GPU and FPGA respectively, the third row shows the disparity maps from semi-global block matching implementation on CPU, GPU and FPGA respectively

The results for the matching shows the local block matching giving better results for processing edges, but the semi-global block implementation giving more matches and a smoother output compared to the "disparity holes" in local block matching, however, it gives more false positives which can be noticed in the top left corner and the bottom area for the disparity map.
It can be noted the GPU semi-global block matching implementation failed to process the nearest parts of the environment (bottom area of disparity map) which would give the highest disparity values, which is due to the libsgm implementation being limited to a maximum disparity of 128.

7.3 Middlebury dataset

Since the ground truth disparity values for the I3DS dataset aren't known. The accuracy of the implementations can't be measured. The 2014 Middlebury dataset [16] was used for further testing of the stereo pipeline implementation by scaling down the stereo pairs into 1920x1080 images, providing a benchmark for the accuracy of the disparity estimation.

7.4 Stereo matching accuracy

The stereo matching algorithms are benchmarked using the 2014 Middlebury dataset, the imagePairReader class was changed to work with this dataset, and every architecture, an image

writing method was added to store the origin of the disparity map, so that the out disparity maps can later be compared to the ground truth disparity maps.
The local block matching tends to perform poorly in areas with little to none textures like the walls in 41, but does perform better in areas with high texture such as edge and corners.

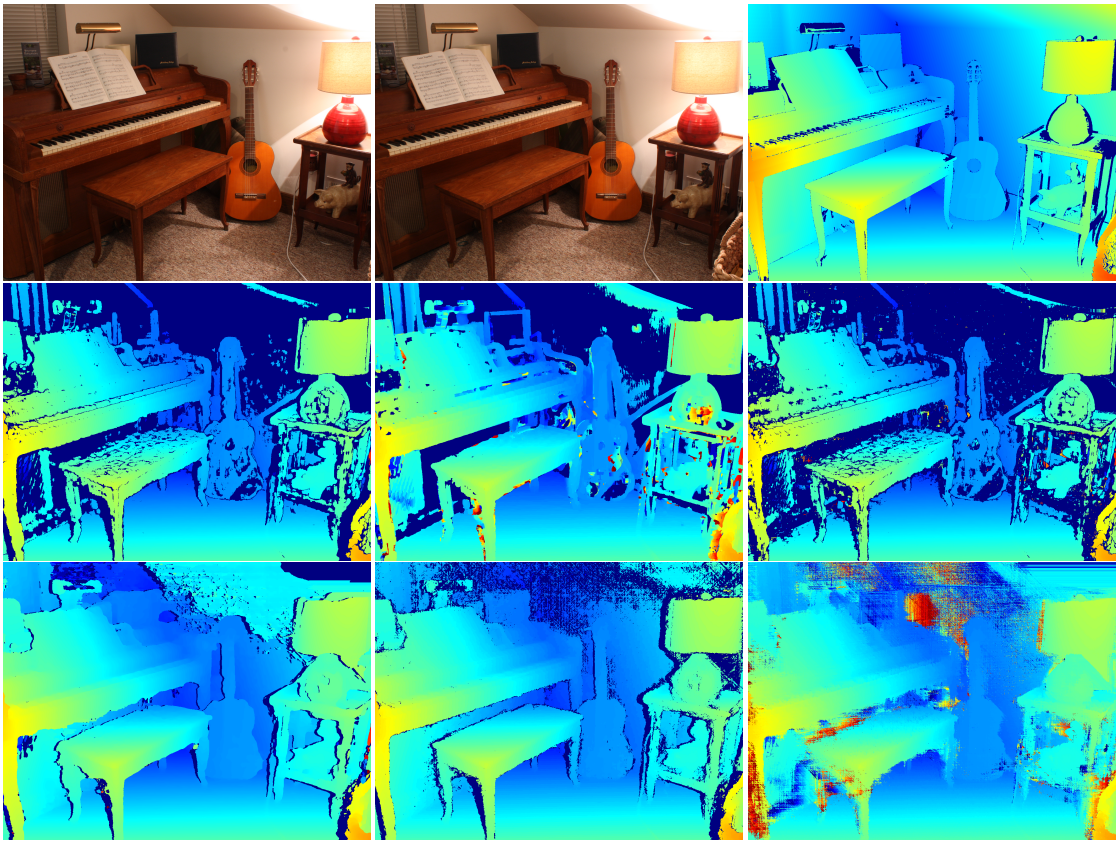


Figure 41: The first row shows a stereo pair left and right images, and the ground truth disparity map respectively from the Middlebury dataset, second row shows the disparity maps from local block matching implementations on CPU, GPU and FPGA respectively, the third row shows the disparity maps from semi-global block matching implementation on CPU, GPU and FPGA respectively

After the dataset was processed with both algorithms and every architecture, an algorithm was developed to loop through every pixel in every image, and compares the pixel value to the corresponding value in the ground truth image, the disparity map was allowed ± 10 pixels range of error to be considered a correct pixel, and then the algorithm will output the percentage of correct disparity values in every disparity map, and the average of every implementation 42.

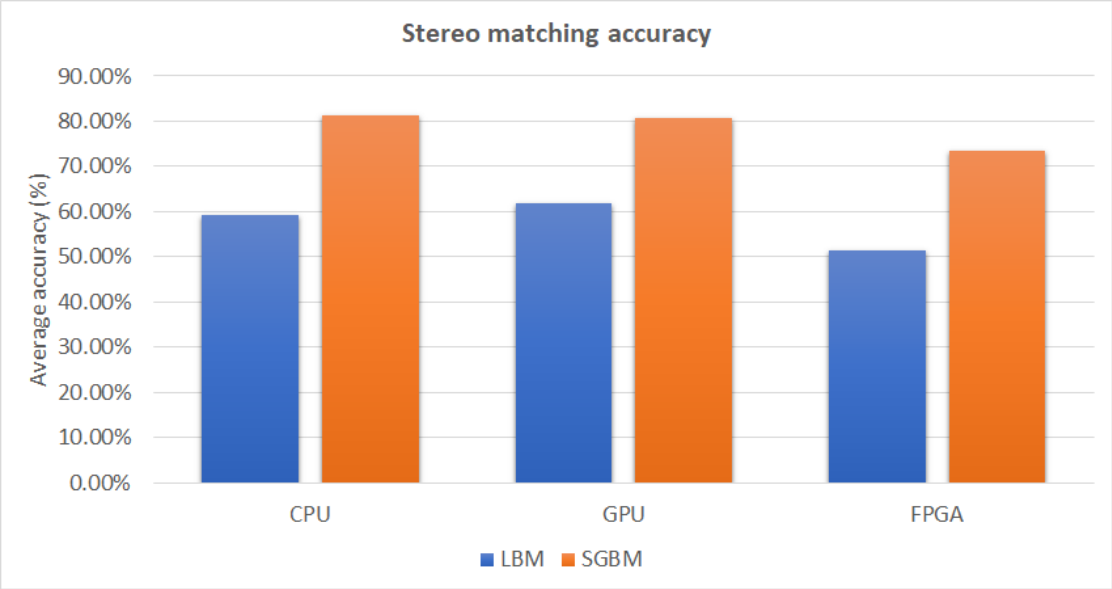


Figure 42: The average accuracy of the implementation of local block matching and semi global block matching tested with the 2014 Middlebury dataset implemented on CPU, GPU and FPGA architectures

7.5 Power consumption

7.5.1 Nvidia Jetson TX2

The Nvidia Jetson TX2 module includes an onboard power monitor, INA3221, to monitor the voltage and current for the power rails. The monitor can read up to three rails, which is enough to read the power consumed by the CPU, GPU and the whole system on chip.
For the TX2 board, the power rails can be read from the following locations using i2c.

```
CPU_monitor = open("/sys/devices/3160000.i2c/i2c-0/
0-0041/iio:device1/in_power1_input", O_RDONLY | O_NONBLOCK);
```

```
GPU_monitor = open("/sys/devices/3160000.i2c/i2c-0/
0-0040/iio:device0/in_power0_input", O_RDONLY | O_NONBLOCK);
```

```
Vin_monitor = open("/sys/devices/3160000.i2c/i2c-0/
0-0041/iio:device1/in_power0_input", O_RDONLY | O_NONBLOCK);
```

The power monitors are set to read in read-only and non-blocking modes as to minimize the impact on the stereo algorithm being tested, a library was implemented to read the power consumption values during a benchmark test, and output the average power consumption values during the final report.

7.5.2 FPGA power consumption

The trenz board doesn't offer a system monitor for the power rails supply to the TE0808-04-09EG-2IE chip, so the power consumption of the FPGA implementations was estimated using the

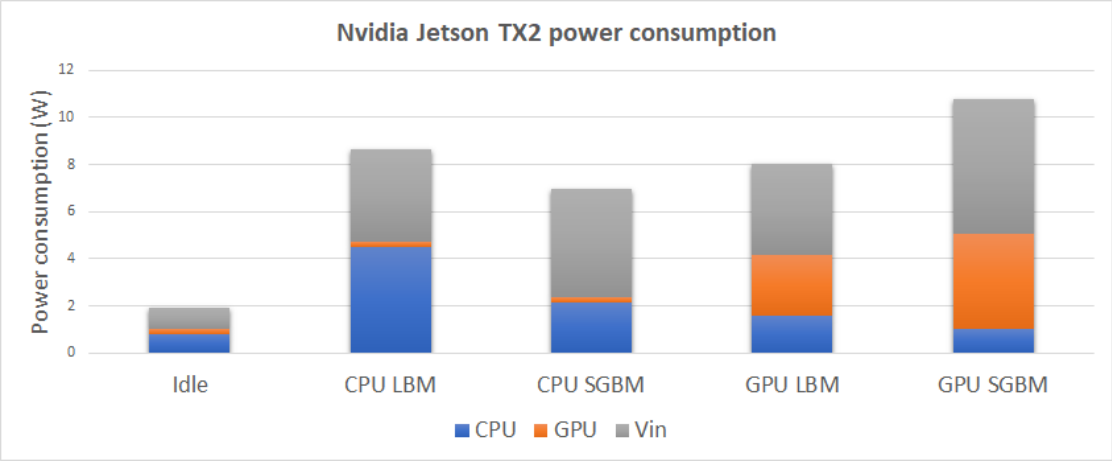


Figure 43: The average power consumption for the stereo pipeline implementations on a 1920x1080p stereo pair image stream, showing the CPU power consumption (blue) GPU power consumption (orange) and the grey area shows the power consumption from all the modules in the SoC such as the DDR power consumption

Vivado power tools and the Xilinx Power Estimator.
The platform design was imported in Vivado, from there the power consumption report was generated however since it is an estimation and shows low confidence 44 it should be taken with a grain of salt.

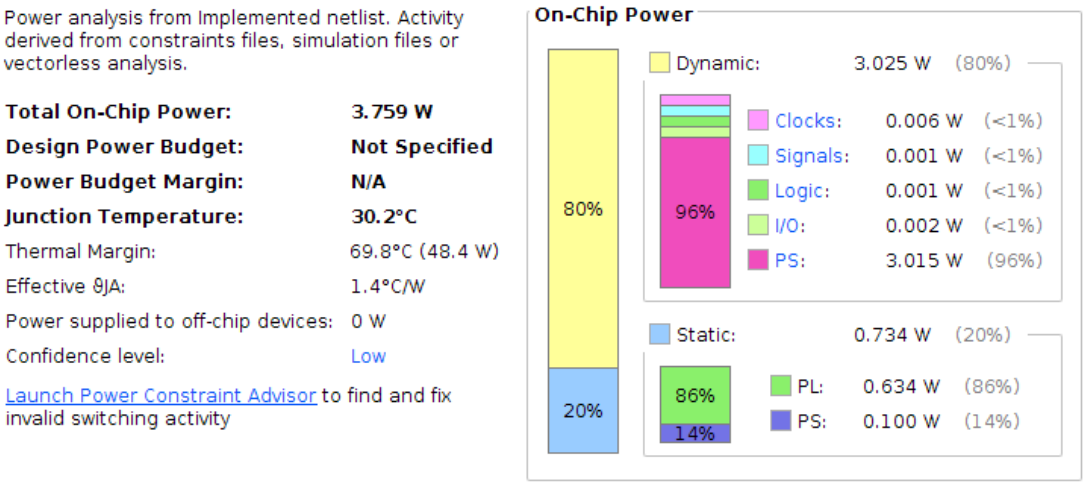


Figure 44: Power consumption of the implementation's platform generated using Vivado tools

The resource usage of the stereo pipeline implementations was exported into the Xilinx Power Estimators, and the power consumption was generated using this tool, it should be noted that the confidence level for this tool as well as low, however for both of these tools the power

consumption results are conservative, so the real power consumption should be significantly lower.

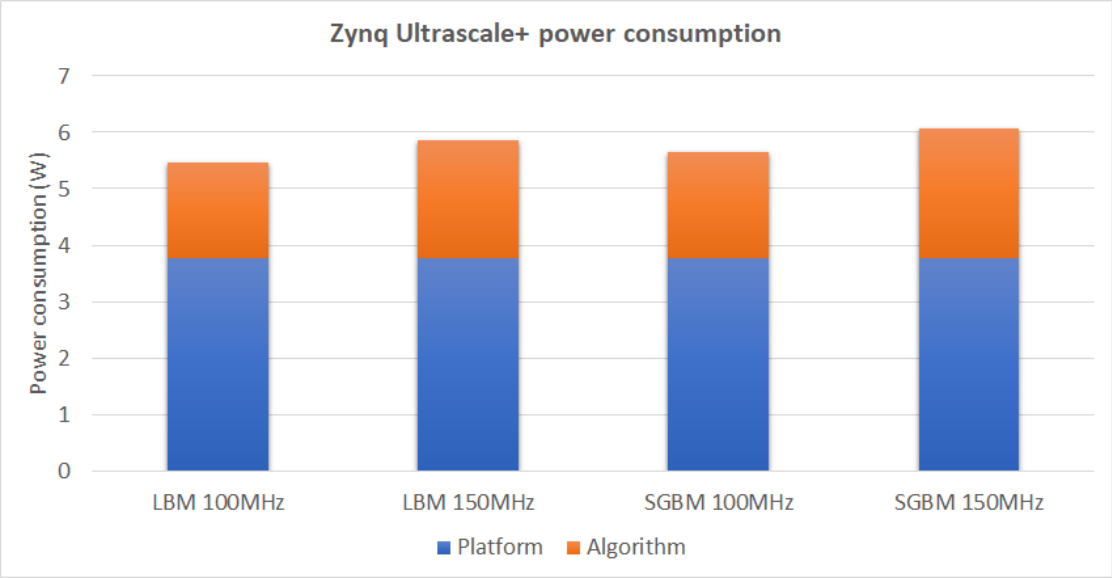


Figure 45: Power consumed by the FPGA implementation of the stereo pipeline on a 1920x1080 stereo pair image stream using both local block matching and semi-global block matching, running at a frequency of 100 MHz and 150 MHz

The stereo pipeline implementations were developed in both 100 Hz and 150 Hz, to choose an optimal implementation with power consumption vs processing time, however, due to the 150 Hz didn't show a significant increase in power consumption 45, the 150 Hz implementation was chosen to be more optimal.
The local block matching and semi-global matching implementations compare differently on each architecture, that is due to the different implementations on each library have different levels of resource usage, directly impacting the power consumption.
Even though the GPU shows a higher power consumption than the CPU, the GPU was able to process a higher number of frames per second 46, so in term of energy consumed per frame the GPU makes up for a more power efficient system.
The FPGA shows the lowest power consumption, that is mainly due to it running on a much lower clock frequency of 150Hz when compared to the CPU running at 2GHz and the GPU running at 1.3GHz, it also does fixed point calculations which do provide higher efficiency.

7.6 Processing time / Frames per second

The processing time is an important factor to be able to achieve real-time constraints. The maximum frames per second are $1/t$ where t is the processing time.

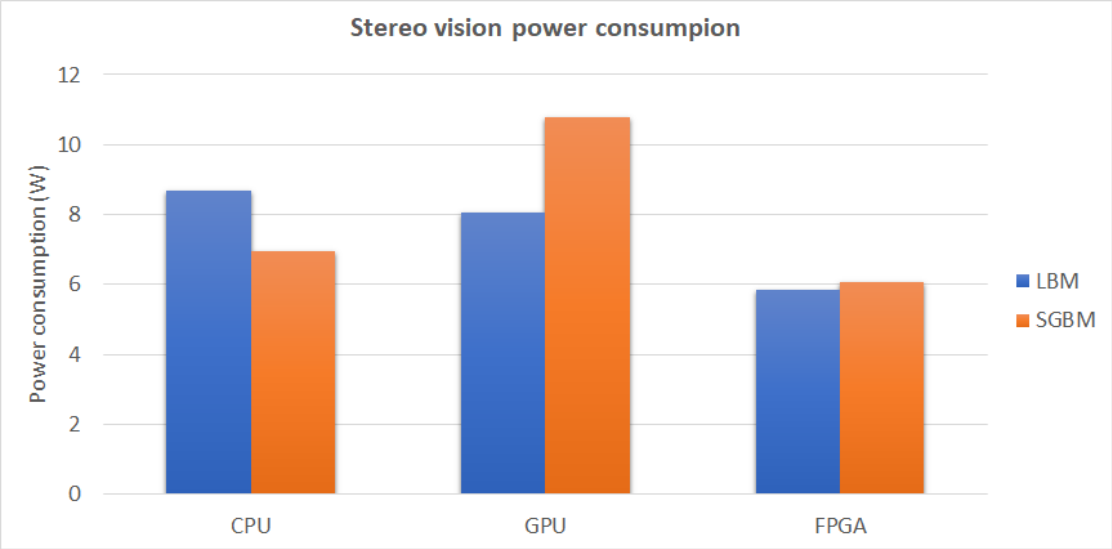


Figure 46: Power consumed for the stereo pipeline implementations LBM and SGBM on a 1920x1080 stereo pair image stream, using CPU, GPU and FPGA architectures

```
-------------------------------------------------------------------------------
Benchmark                                      Time           CPU   Iterations
-------------------------------------------------------------------------------
BM_GPU_LBM/min_time:30.000/manual_time       90.7 ms         176 ms          463
 Average CPU power Consumed: 1571.28mW
 Average GPU power Consumed: 2578.45mW
 Average Vin power Consumed: 8042.22mW
BM_GPU_SGBM/min_time:30.000/manual_time       185 ms         183 ms          227
 Average CPU power Consumed: 1020.14mW
 Average GPU power Consumed: 4060.18mW
 Average Vin power Consumed: 10793.5mW
BM_CPU_LBM/min_time:30.000/manual_time        206 ms         354 ms          205
 Average CPU power Consumed: 4513.41mW
 Average GPU power Consumed: 229mW
 Average Vin power Consumed: 8670.15mW
BM_CPU_SGBM3/min_time:30.000/manual_time      3245 ms        3369 ms           13
 Average CPU power Consumed: 2149.85mW
 Average GPU power Consumed: 229mW
 Average Vin power Consumed: 6944.31mW
```

Figure 47: Google benchmark report with integrated TX2 power measurements

The google benchmark library was used to calculate the average processing time of every implementation on both the i3ds and Middlebury datasets. However, the default clock timer didn't provide accurate results when benchmarking GPU accelerated algorithm so, a manual clock was used, performed using the Chrono library, the following code snippet shows an example for the use of google benchmark, with the GPU, accelerated local block matching.

```
#include <benchmark/benchmark.h>

static void BM_GPU_LBM(benchmark::State &state)
```

```
{

        TX2PowerMonitor pmon{state};
        stereoPipeline stereo;
        imagePairReader images("image_list.xml", "gl");
        cv::Mat imgL, imgR, DisparityMap;

        images.randomStart();
        images.readNext(&imgL, &imgR);

        stereo.CPU_getCameraParameters();
        stereo.CPU_computeRemapMatrix(&imgL, &imgR);
        stereo.GPU_initLBM(15, 160);


        for (auto _ : state)
        {

                images.readNext(&imgL, &imgR);
                auto start = std::chrono::high_resolution_clock::now();

                DisparityMap = stereo.GPU_LBM(&imgL, &imgR);

                auto end = std::chrono::high_resolution_clock::now();
                auto elapsed_seconds = std::chrono::duration_cast
                    ↪ <std::chrono::duration<double>>(end - start);
                state.SetIterationTime(elapsed_seconds.count());
                pmon.measurePower();
                images.writeImg(DisparityMap);
        }
        pmon.reportAverage(state);
}

BENCHMARK(BM_GPU_LBM)->UseManualTime()
->Unit(benchmark::kMillisecond)->MinTime(30);

BENCHMARK_MAIN();
```

The TX2 power monitor shown earlier was integrated with the Google benchmark, also providing the average power consumption of the stereo pipeline alongside the processing time 47. The local block matching method shows half the processing time compared to the semi-global based approach, in the GPU and FPGA implementation. The FPGA based implementation shows a significant speedup compared to both CPU and GPU based approaches 6x as fast as the CPU implementation and 3x as fast the GPU implementation.

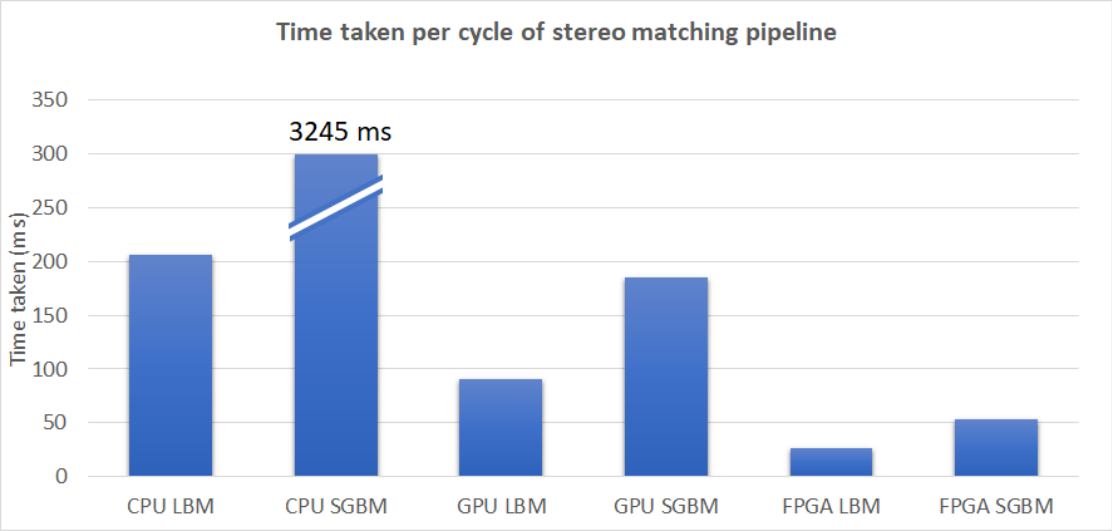


Figure 48: Chart showing the average processing time for every implementation with CPU LBM at 206ms, CPU SGBM at 3245 ms, GPU LBM at 91 ms, GPU SGBM at 185 ms, FPGA LBM at 26 ms and FPGA SGBM at 53 ms

8 Discussion

The stereo pipeline was implemented using OpenCV on the CPU side, and for the GPU the OpenCV::CUDA library was used for the local block matching, and libsgm for the semi-global block matching implementation. Finally the pipeline was synthesized for the FPGA using the XFopenCV libraries.
The stereo pipeline implementations were tested on the I3DS dataset to test for correct real-time undistortion and rectification before the stereo matching, and for the GPU and FPGA implementations, no frame rates are lost due to pipelining.
The stereo matching accuracy was tested using the Middlebury dataset to show the robustness of the algorithm in different environments and to benchmark the accuracy.
The aim of these benchmarks and tests to find an optimal stereo system implementation, and to achieve optimality. The following factors need to be taken into consideration; development costs, stereo matching accuracy, frame rates and power consumption.

8.0.1 Developement costs

The use of open source libraries guarantees development on a higher abstraction, significantly reducing development costs and time, and time to market, however, it impairs the developer's ability to take full advantage of the underlying hardware.
This has been the main advantage for the use of CPU and GPU over FPGAs in vision applications with the OpenCV and CUDA libraries, however the use of high-level synthesis, HW/SW co-design with the SDx tool, and now the XFopenCV library developed by Xilinx promised to bring FPGA development time/costs closer to the CPU and GPU.
However, even with the use of these tools, the development time/costs on the FPGA remains significantly larger (in the case of this project more than 10x the development time of CPU/GPU). Mainly because these tools are relatively new, which means dealing with multiple bugs in the toolchain. High-level synthesis takes a very long time; compiling a small change in the design could take over 10 hours to synthesize, and might not even complete if there is an error. Debugging is very difficult due to the lack of descriptiveness from the errors, and finally working with a custom hardware board could require a custom hardware platform A being built which requires a great deal of low-level hardware knowledge of the board, and a software platform B (custom operating system).

8.0.2 Stereo matching accuracy

The local block matching method was more accurate at matching areas with large changes (edges and corners) however semi-global block matching method was better at matching areas with low textures (walls), however, due to the LBM method usually leaving holes for the areas with low confidence, SGBM scored an added 20% over LBM's scores for the Middlebury matching accuracy 49.

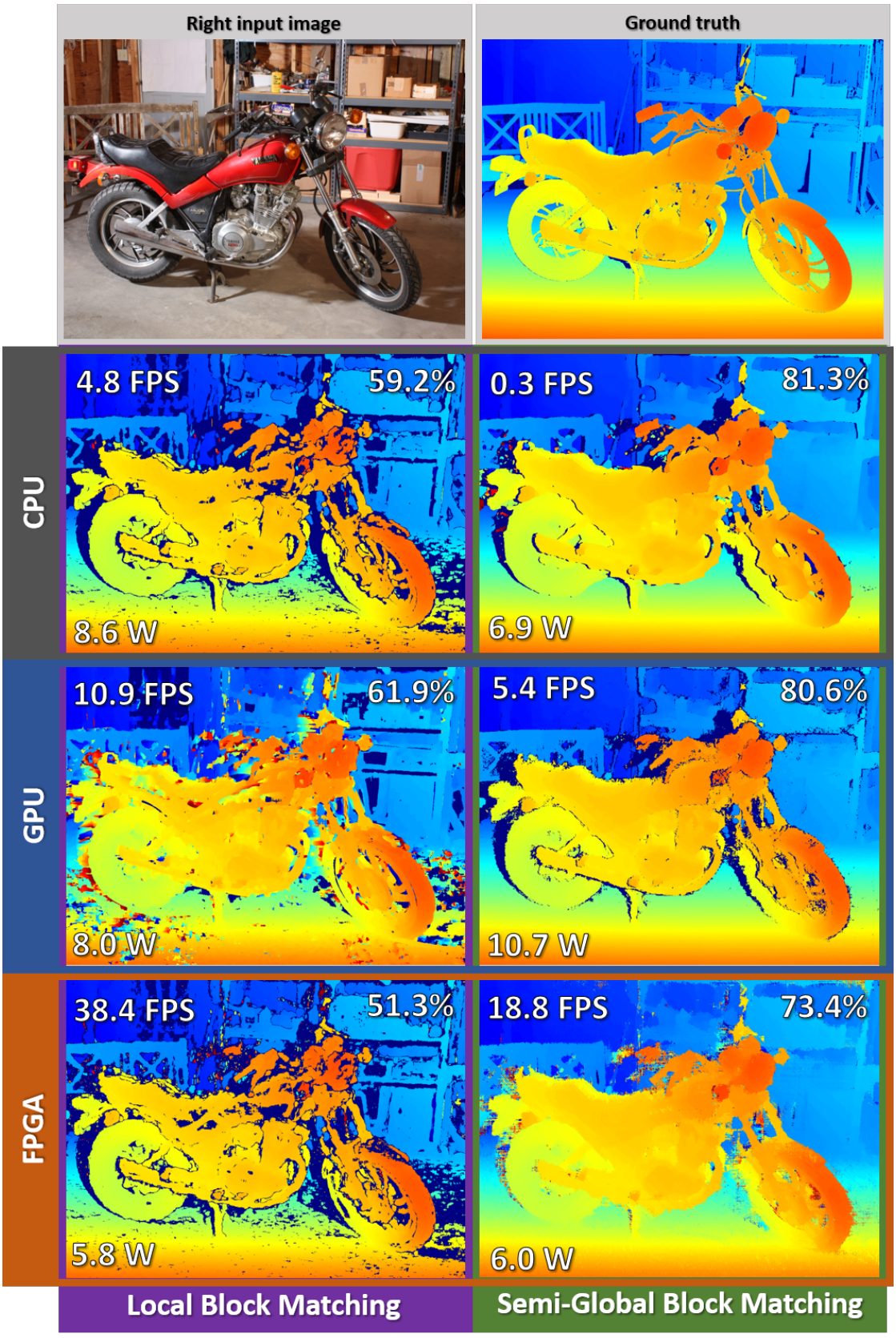


Figure 49: The output disparity maps from stereo matching pipeline implementations on the 2014 Middlebury dataset, 1920x1080 images, 160 max disparity, using local and semi-global block matching, implemented on an embedded CPU (ARM Cortex-A57), embedded GPU (NVIDIA Pascal) and FPGA (Xilinx XCZU9EG). For every output disparity map the average frames per second are shown on the top left corner, the average accuracy on the top right and the average system on chip power consumption on the bottom left

The CPU and GPU implementations scored similarly and 10% higher than the FPGA implementation, mainly because the CPU and GPU use floating point arithmetics while the FPGA uses fixed point arithmetics.

8.0.3 Frame rates

The stereo matching application is a so-called "embarrassingly parallel" application, due to the CPU's nature of serial computation, it can't take advantage of that, which explains why the embedded CPU performed so much lower than the GPU and FPGA implementations.
FPGAs allow for a much higher degree of freedom when designing the computational blocks for the stereo matching algorithm, unlike the fixed computational cores in the GPU, combined with the fixed point arithmetic, FPGAs scored 4x higher than the GPU on frame rates.

8.0.4 Power consumption

The GPU scored similarly to the CPU for the LBM approach, and larger on the SGBM approach, but even with the larger power consumption, the GPU implementation consumes significantly less energy per computed frame, which make the GPU more power efficient than the CPU implementation.
The FPGA power consumption scores are estimated not measured, the FPGA consumed the least power, which makes the FPGA by far the most efficient architecture when it comes to energy consumption per frame, one of the biggest reasons for this is that the FPGA implementation runs 150 MHz frequency, while the CPU runs on 2GHz and the GPU 1.2 GHz.

8.1 Limitations

A major limitation of this report is that the trenz board TE0808, doesn't offer an accessible system monitor during operation, which led to the usage of Vivado and Xilinx Power Estimators, to estimate the power usage using the design developed, the resulting power estimates are in low confidence.
For the GPU the VisionWorks library is reported to outperform the CUDA OpenCV implementation, however, for this report, it was important to minimize the variance with the implementation of the algorithm, to ensure a more accurate representation of the architectural differences across all hardware.

9 Conclusion

The project explored different implementations of stereo systems, local-based approach and semi-global approach, in all architectures the semi-global approach was more accurate, while the local-based approach was faster.
In the implementation chapter; A stereo system was developed for embedded devices using different vision libraries based on OpenCV. Different optimization techniques were employed to take full advantage of the underlying hardware.
The OpenCV implementation on CPU provided the highest accuracy for local-block matching, while libsgm implementation on GPU provided the highest accuracy for semi-global block matching, while the XFopenCV implementation on the FPGA scored the lowest.
The FPGA implementation provided the highest embedded performance; in terms of frame rates and energy consumed per frame. GPU scored mid-way, while the CPU had the lowest performance.
Finally, the optimal configuration of an embedded stereo system depends on the application; if low-power real-time performance is critical, the FPGA implementations provides the best performance. However, the GPU still acceptable performance, but due to its floating point operations, it doesn't sacrifice accuracy, which can then be used for applications where accuracy is of higher importance.

9.1 Further work

The project explored different implementations of stereo systems, on different architectures. The project lays a strong foundation for embedded stereo development especially for projects that aim for high performance with low development costs.
This thesis not only takes into account performance in frame rates and accuracy but also takes into account development costs. However, in applications such as planetary explorations and orbital debris removal, the development costs are minimal as compared to the overall costs, in such cases, it would make sense to develop without the use of libraries. Designing customized hardware accelerators takes full advantage of the underlying hardware, and it would also allow for further optimizations of the stereo pipeline algorithm for the planetary exploration I3DS dataset.

A Building the hardware platform

The SDSoC platform needs the hardware platform to be configured to enable Hardware/Software co-design, for a custom board like the Trenz SK0808 board and the Ultrazed-3EG used in this project require a custom hardware platform to be used, it is advised to use the platform supplied by the manufacturer of the board, as building the platform is very time consuming and requires deep understanding of the underlying hardware.
The hardware platform defines the processor type and configures the memory, logical and physical interfaces, that are to be used by the functions accelerated in hardware, and is built using the Vivado software tools provided by Xilinx. for more details refer to the Xilinx [31] guide.

A.1 Hardware block design

The hardware platform is created in Vivado, and a new hardware block design is generated, The design requires some rules to be followed [31], The following design enables SDSoC development, and hardware accelerated vision applications.

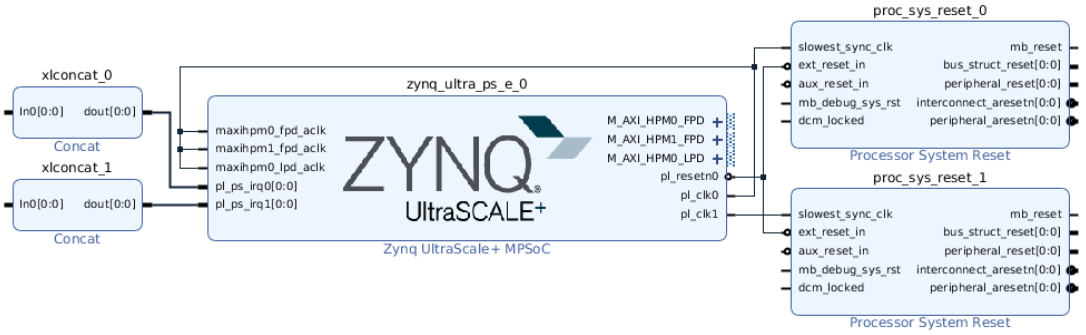


Figure 50: Minimal hardware platform block design for SDSoC development

- **The embedded processor IP**
  - **IRQ[0-7]** are enabled to allow for PL to PS interrupts
  - **AXI HMP0 FPD** and **AXI HMP1 FPD** are disabled to allow SDSoC to use them for the PS to PL interrupts
- **PL Clocks** customized to generate at least 2 PL clocks at 100 MHz and 200 MHz clock frequency, with the 100 MHz clock as the default clock
- **Processor system reset** each reset connected to a different clock frequency

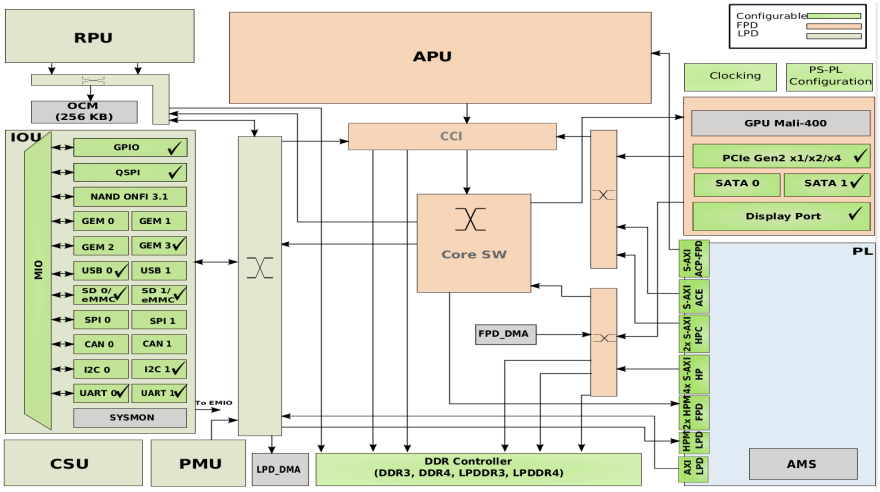


Figure 51: High-level system view with all the active peripherals highlighted

- **Concat** to concatenate bus signals of varying widths

For the SK0808 board, trenz electronics [8] provides TCL scripts that includes the configurations for all the peripherals within the board, latest version can be downloaded from the website [32].

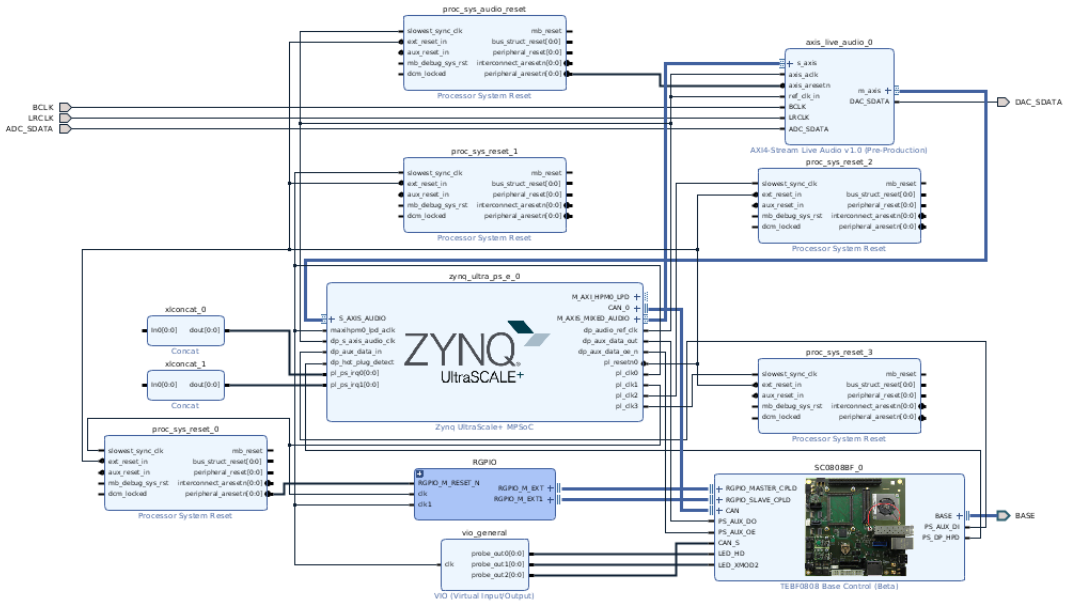


Figure 52: SK0808 board hardware platform block design

Functionality for multiple features were added for I/O devices including the display port controller, audio, push button, switches and LEDs.

A.2 Configure platform and interface properties

The platform interfaces need to be declared before synthesis, so it can be used by the SDSoC compiler, and it can be achieved by setting the PFM properties on the interface ports as seen in the figure 53.

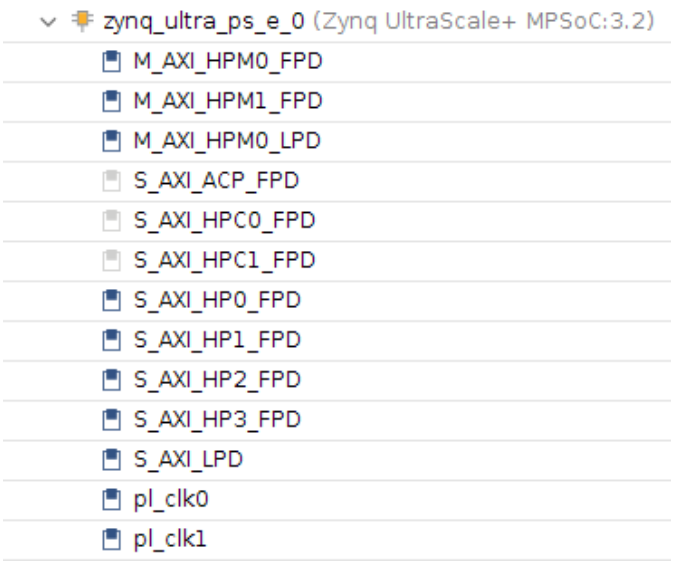


Figure 53: Interface properties for an SDSoC hardware platform

it's important to remember to set the name of the platform same as the project name otherwise there will be errors, and the 100 MHz clock has to be set as the default clock.

A.3 Synthesizing the hardware design

Now the design is complete, and can be verified by right clicking the block design and choosing "Validate design", next generate the output products, and create and HDL wrapper, so that the process of generating the bitstream can be started (Synthesis, Implementation, Bitstream generation).

A.3.1 Generating the DSA file

The DSA file can be generated by switching to the TCL console and using the write dsa command to generate the file.
Finally both the ***.dsa** file and the FPGA bitstream ***.bit** are the files we need to proceed to the next step in the SDSoC platform generation.

A.3.2 Note: System format bug

A bug was encountered in Vivado, causing synthesis errors. which is fixed by changing the system's format to US format (not norwegian or german)

B Building the operating system

The second level of the SDSoC platform is building the bootloaders and the operating system, in this project a linux operating system and the bootloaders are built using the Petalinux tools.

B.1 Petalinux tools

Petalinux is provided by Xilinx and is based on the Yocto project, and it can be used to generate the bootloader, linux kernel and all the user space libraries. for the SDSoC platform two files are needed

- **boot.bin** the bootloader file; which contains the first stage bootloader $FSBL.elf$, the FPGA bitstream $*.bit$ the U-Boot bootloader $*.elf$
- **image.ub** the linux image; which contains the device tree block $*.dtb$, the ram-disk image $*.gz$ and the linux kernel.

These files are also needed to boot from an SD card, both of these files need to be uploaded to the FAT32 partition labeled **BOOT** and for the other partition **ROOTFS** is where the root file system will be stored.

B.2 Building the PetaLinux Image

After downloading and installing the Petalinux tools, with the same version as your SDSoC installation.
first step is to create a Petalinux project.

```
petalinux-create -t project --template zynqMP -n <project-name>
cd <project-name>
```

B.3 Petalinux hardware description file

The petalinux project will be configured using the DSA generated from the Vivado tools.

```
petalinux-config --get-hw-description= <DSA path>
```

First the boot arguments need to be added.

```
DTG Setting -> generate boot args automatically (OFF)
Kernel Bootargs -> user set kernel bootargs
```

Arguments are added to run the hardware accelerated functions correctly and so that the SD card can properly load the root file system from the second partition **ROOTFS** If you would like to remove the unimportant message during the booting process change boot args to include "quiet" at the end of the boot arguments.

```
setenv bootargs 'earlycon clk_ignore_unused root=/dev/mmcblk1p2 rw
    ↪ rootfstype=ext4 rootwait quiet'
```

B.4 PetaLinux kernel

Next step is to configure the linux kernel

```
petalinux-config -c kernel
```

Some settings need to be changed for the SDSoC application to run properly the CMA size needs be larger for the SDS-alloc buffers

```
Device Drivers -> Generic Driver Options -> Size in Megabytes(1024)
```

Enable staging drivers

```
Device Drivers -> Staging drivers (ON)
```

Enable APF management driver:

```
Device Drivers -> Staging drivers -> Xilinx APF Accelerator driver (ON)
```

Enable APF DMA driver:

```
Device Drivers -> Staging drivers -> Xilinx APF Accelerator driver -> Xilinx
    ↪ APF DMA engines support (ON)
```

The CPU idle and frequency scaling must be turned off.

```
CPU Power Management -> CPU idle -> CPU idle PM support (OFF)
CPU Power Management -> CPU Frequency scaling -> CPU Frequency scaling (OFF)
```

B.5 Configure petalinux rootfs

Now the root file system needs to be configured.

```
petalinux-config -c rootfs
```

Add the stdc++ libraries

```
Filesystem Packages -> misc -> gcc-runtime -> libstdc++ (ON)
```

Add the openCV libraries

```
Add packagegroup-petalinux-opencv
```

B.6 Add device tree fragment for APF driver

The $system - user.dtsi$ file needs to be edited

```
sudo gedit
    ↪ project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi
```

Copy the following code snippet into the dtsi file, it includes system information the trenz board, disable write protection for the SD card, and enables SDSoC applications.

```
/include/ "system-conf.dtsi"
 /{
  };
&gem3 {
   status = "okay";
   local-mac-address = [00 0a 35 00 02 90];
   phy-mode = "rgmii-id";
   phy-handle = <&phy0>;
   phy0: phy@9 {
          reg = <0x9>;
          ti,rx-internal-delay = <0x5>;
          ti,tx-internal-delay = <0x5>;
          ti,fifo-depth = <0x1>;
   };
};

&i2c1 {
   status = "okay";
   clock-frequency = <400000>;

   i2cswitch@70 { /* U7 on UZ3EG SOM */
          compatible = "nxp,pca9542";
          #address-cells = <1>;
          #size-cells = <0>;
          reg = <0x70>;
          i2c@0 { /* i2c mw 70 0 1 */
                 #address-cells = <1>;
                 #size-cells = <0>;
                 reg = <0>;
                 /* IIC_EEPROM */
                 eeprom@51 { /* U5 on UZ3EG IOCC and U7 on the UZ7EV EVCC*/
                        compatible = "at,24c08";
                        reg = <0x51>;
                 };
          };
   };
};

&qspi {
   #address-cells = <1>;
   #size-cells = <0>;
   status = "okay";
   is-dual = <1>; /* Set for dual-parallel QSPI config */
   num-cs = <2>;
   xlnx,fb-clk = <0x1>;
   flash0: flash@0 {
```

```
            /* The Flash described below doesn't match our board
                ↪ ("micron,n25qu256a"), but is needed */
            /* so the Flash MTD partitions are correctly identified in
                ↪ /proc/mtd */
               compatible = "micron,m25p80"; /* 32MB */
               #address-cells = <1>;
               #size-cells = <1>;
               reg = <0x0>;
               spi-tx-bus-width = <1>;
               spi-rx-bus-width = <4>; /* FIXME also DUAL configuration
                   ↪ possible */
               spi-max-frequency = <108000000>; /* Set to 108000000 Based on
                   ↪ DC1 spec */
      };
   };

   /* SD0 eMMC, 8-bit wide data bus */
   &sdhci0 {
      status = "okay";
      bus-width = <8>;
      max-frequency = <50000000>;
   };

   /* SD1 with level shifter */
   &sdhci1 {
      status = "okay";
      max-frequency = <50000000>;
      no-1-8-v; /* for 1.0 silicon */
      disable-wp; /* Disable Write protection in SD card */
   };

   /* ULPI SMSC USB3320 */
   &usb0 {
      status = "okay";
   };

   &dwc3_0 {
      status = "okay";
      dr_mode = "host";
      phy-names = "usb3-phy";
   };
   /* APF driver for SDx */
   /{
   xlnk {
   compatible = "xlnx,xlnk-1.0";
   };
   };
```

B.7 Building the PetaLinux image

Finally now the petalinux image can built by using the following command from within the root directory of the petalinux project.

```
petalinux-build
```

B.8 Generate boot files

The boot.bin file can be generated using the following command, make sure the bitstream file built in the hardware platform, is included in the images/linux/ folder.

```
petalinux-package --boot --format BIN --fsbl images/linux/zynqmp_fsbl.elf
    ↪ --u-boot images/linux/u-boot.elf --pmufw images/linux/pmufw.elf --fpga
    ↪ images/linux/*.bit --force
```

B.9 Building the SYSROOT folder

The following command downloads all the libraries that will be used in the project.

```
petalinux-build --sdk
```

The SYSROOT folder is built within the images/linux folder this file includes all the user space libraries that will be used, in the SDSoC project this folder will be referenced as the "SYSROOT" folder, and the contents will be copied in the **ROOTFS** partition in the SD card

```
petalinux-package --sysroot
```

B.10 The petalinux image

Go to the directory $images/linux/$

```
cd images/linux
```

The files used in the boot process need to be copied in a folder labeled boot while the image.ub file is copied into a folder labeled image.

```
mkdir ./boot
mkdir ./image
cp u-boot.elf ./boot/u-boot.elf
cp *fsbl.elf ./boot/fsbl.elf
cp bl31.elf ./boot/bl31.elf
cp pmufw.elf ./boot/pmufw.elf
cp image.ub ./image/image.ub
sudo gedit boot.bif
```

Create a boot image format file, which is used to compile the contents of the boot folder into a BOOT.BIN file from the SDSoC platform creation utility.

```
the_ROM_image:
{
```

```
  [fsbl_config] a53_x64
  [bootloader]<fsbl.elf>
  [pmufw_image]<pmufw.elf>
  [destination_device=pl] <bitstream>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <u-boot.elf>
}
```

Now the petalinux project is complete, and an SDSoC platform can be created from the boot files, linux image file and root filesystem from this project and the DSA file from the hardware platform.

Bibliography

[1] Cardoso, J. M., Coutinho, J. G. F., & Diniz, P. C. 2017. Chapter 2 - high-performance embedded computing. In *Embedded Computing for High Performance*, Cardoso, J. M., Coutinho, J. G. F., & Diniz, P. C., eds, 17 – 56. Morgan Kaufmann, Boston. URL: http://www.sciencedirect.com/science/article/pii/B9780128041895000028, doi:https://doi.org/10.1016/B978-0-12-804189-5.00002-8.

[2] NVIDIA. 2019. NVIDIA tesla p100. URL: https://www.xilinx.com/support/documentation/sw{_}manuals/petalinux2013{_}10/ug976-petalinux-installation.pdf.

[3] Xilinx. 2018. UltraScale Architecture DSP Slice (UG579). URL: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[4] Navigation, V.-b. 2018. High-Performance Embedded Computing in Space : Evaluation of Platforms for High Performance Embedded Computing in Space : Evaluation of Platforms for Vision-based Navigation. (February). doi:10.2514/1.I010555.

[5] OpenCV. OpenCV camera calibration and 3d reconstruction. URL: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[6] Opsahl, T. Machine view, lecture 6.1: Basic epipolar geometry. URL: https://www.uio.no/studier/emner/matnat/its/UNIK4690/v16/forelesninger/lecture_6_1_basic_epipolar-geometry.pdf.

[7] Avnet. Ultrazed-EG starter kit. URL: http://www.zedboard.org/product/ultrazed-eg-starter-kit.

[8] Trenz-electronics. TE0808-04-09-2IE-S starter kit. URL: https://shop.trenz-electronic.de/en/TE0808-04-09-2IE-S-TE0808-04-09-2IE-S-Starter-Kit.

[9] NVIDIA. NVIDIA jetson getting started. URL: https://developer.nvidia.com/embedded/learn/getting-started-jetson.

[10] Forshaw, J. L., Aglietti, G. S., Navarathinam, N., Kadhem, H., Salmon, T., Pisseloup, A., Joffre, E., Chabot, T., Retat, I., Axthelm, R., et al. 2016. Removedebris: An in-orbit active debris removal demonstration mission. *Acta Astronautica*, 127, 448–463.

[11] McLennan, S. M. & McSween, H. Y. April 2018. Recent Accomplishments in Mars Exploration: The Rover Perspective. In *Second International Mars Sample Return*, volume 2071 of *LPI Contributions*, 6034.

[12] Peña Carrillo, D. A. *Efficient stereo matching and obstacle detection using edges in images from a moving vehicle*. PhD thesis, Dublin City University, 2017.

[13] Lentaris, G., Maragos, K., Stratakos, I., Papadopoulos, L., Papanikolaou, O., Soudris, D., Lourakis, M., Zabulis, X., Gonzalez-Arjona, D., & Furano, G. 02 2018. High-performance embedded computing in space: Evaluation of platforms for vision-based navigation. *Journal of Aerospace Information Systems*, In press. doi:10.2514/1.I010555.

[14] Ewbank, T. Efficient and precise stereoscopic vision for humanoid robots. Master's thesis, University of Liege, 2017.

[15] Lehnert, C. F., English, A., McCool, C., Tow, A. W., & Perez, T. 2017. Autonomous sweet pepper harvesting for protected cropping systems. *IEEE Robotics and Automation Letters*, 2, 872–879.

[16] Scharstein, D. & Szeliski, R. April 2002. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vision*, 47(1-3), 7–42. URL: https://doi.org/10.1023/A:1014573219977, doi:10.1023/A:1014573219977.

[17] Chappuis, T. Mpsoc for high performance heterogeneous computing. Master's thesis, The University of Applied Sciences and Arts of Western Switzerland, 2018.

[18] Perri, S., Frustaci, F., Spagnolo, F., & Corsonello, P. 05 2018. Stereo vision architecture for heterogeneous systems-on-chip. *Journal of Real-Time Image Processing*. doi:10.1007/s11554-018-0782-z.

[19] Hillerstrom, P. Gpu-accelerated real-time stereo matching. Master's thesis, Chalmers University of Technology, 2017.

[20] Nvidia. 2019. Cuda c programming guide. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[21] Mondal, A. & Ali, M. 06 2017. Performance review of the stereo matching algorithms. *American Journal of Computer Science and Information Engineering*, 4, 7–15.

[22] Zicari, P., Perri, S., Corsonello, P., & Cocorullo, G. 2012. Microprocessors and Microsystems Low-cost FPGA stereo vision system for real time disparity maps calculation. 36, 281–288. doi:10.1016/j.micpro.2012.02.014.

[23] Hirschmuller, H. & Scharstein, D. Sep. 2009. Evaluation of stereo matching costs on images with radiometric differences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(9), 1582–1599. doi:10.1109/TPAMI.2008.221.

[24] Zabih, R. & Woodfill, J. 1994. Non-parametric local transforms for computing visual correspondence. In *European conference on computer vision*, 151–158. Springer.

[25] Hirschmuller, H. Feb 2008. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2), 328–341. doi:10.1109/TPAMI.2007.1166.

[26] Xilinx. OpenCV: open source computer vision library. URL: https://github.com/opencv/opencv.

[27] Fixstars. libSGM a cuda implementation performing semi-global matching. URL: https://github.com/fixstars/libSGM.

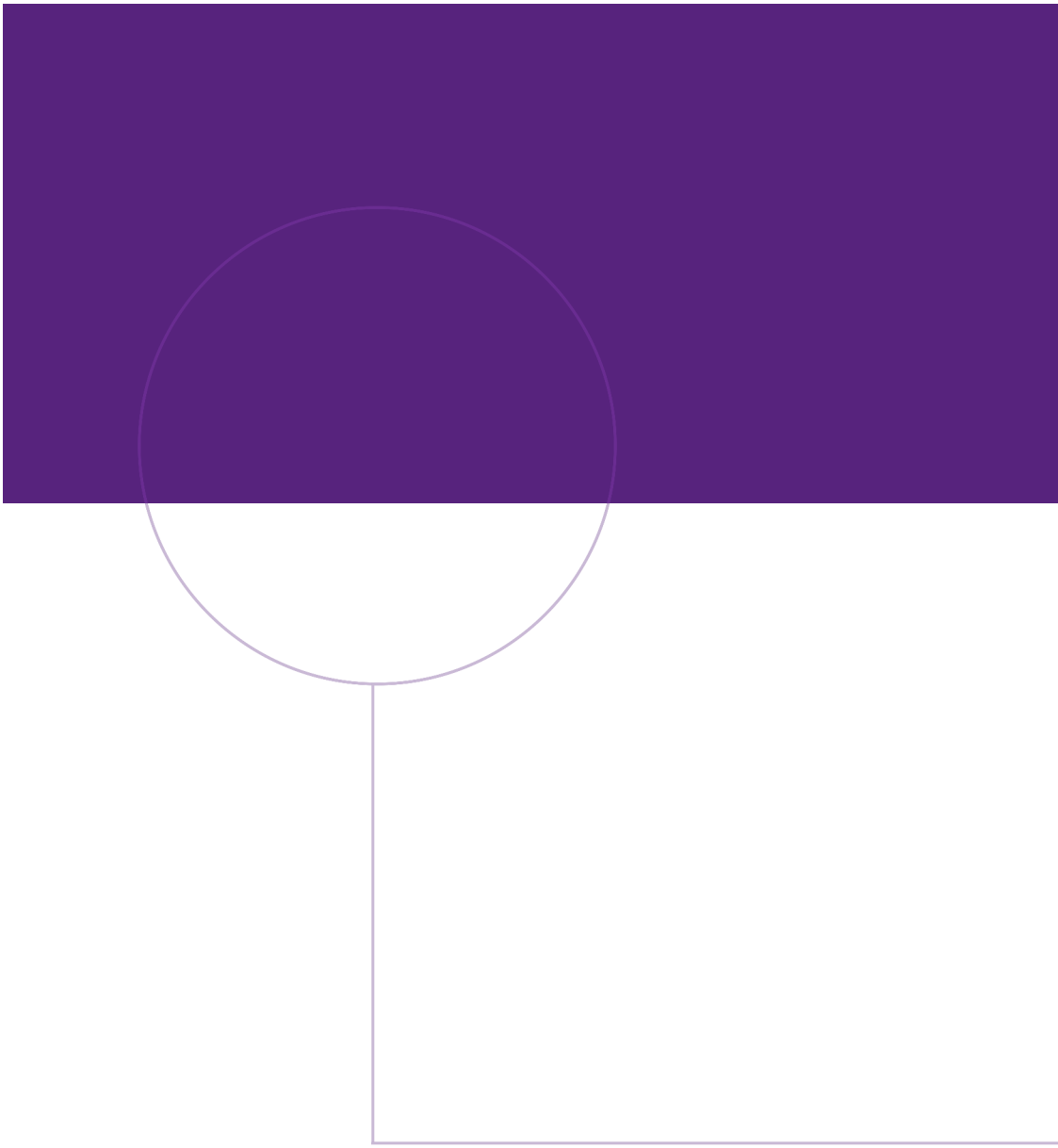
[28] Xilinx. xfOpenCV. URL: https://github.com/Xilinx/xfopencv.

[29] Google. Google Benchmark a library to benchmark code snippets. URL: https://github.com/opencv/opencv.

[30] Zhang, Z. November 2000. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11), 1330–1334. URL: http://dx.doi.org/10.1109/34.888718, doi:10.1109/34.888718.

[31] Xilinx. 2019. Sdsoc environment platform development guideUG1146. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1146-sdsoc-platform-development.pdf.

[32] Trenz-electronics. TE0808-04-09-2IE-S starter kit download page. URL: https://shop.trenz-electronic.de/en/Download/?path=Trenz_Electronic/Modules_and_Module_Carriers/5.2x7.6/TE0808/Reference_Design/2018.2.



NTNU
Norwegian University of
Science and Technology