Master's thesis

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

Maximilian Peter Montzka

Fast Spectrograph Corrections on Programmable Logic

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic Co-Supervisor: Joseph Garrett
July 2020

Norwegian University of Science and Technology

Norwegian University of
Science and Technology

Master Thesis in Electronic Systems Design

Fast Spectrograph Corrections on Programmable Logic

Maximilian Peter Montzka

July 2020

Abstract

This master thesis if written as part of the larger project that is the HYPer-spectral Smallsat for ocean Observation (HYPSO) mission at NTNU. The goal of the Hypso mission is to eventually launch several CubeSats into earths orbit. The CubeSats are then used for oceanographic observation. The CubeSats in question have the dimensions 10 cm × 20 cm × 30 cm. This small form factor limits the amount of hardware which can be put onto these satellites. Meaning software and hardware used on these CubeSats has to be designed around these limitations.

The problem this thesis focuses on is the correction of misregistrations and aberrations recorded by the hyperspectral imager, that is on-board the CubeSats. Specifically the correction of the smile and keystone effects are focused on. Prior to this thesis a feasibility study was conducted, in which a software solution for corrections was developed. This software solution however was not able to do the corrections as fast as desired, thus leading to this thesis, which aims to do the corrections on programmable logic.

The corrections are separated into two main functions. The first one is the generation of the correction matrix. This is done in software developed during the feasibility study, and will not be part of the work done in this thesis. The second function does the actual corrections. The corrections are done by multiplying the correction matrix with an uncorrected hyperspectral frame.

The corrections on programmable logic are done by transferring the correction data from the on-board memory, with the help of the AXI DMA, to the programmable logic. Which then does the corrections and sends the corrected data back to memory, via the AXI DMA. The final solution contains only one thread and performance worse than the software solution. The worse performance is a result of data transfer speeds.

Sammendrag

Denne masteroppgaven er skrevet som en del av HYPer-spectral Smallsat for ocean Observation (HYPSO) prosjektet på NTNU. Målet til HYPSO prosjektet er å sende opp flere såkalte CubeSats til jordas omløpsbane. CubeSat'ene blir så brukt for oseanografisk observasjon. CubeSat'ene har dimensjonene 10 cm × 20 cm × 30 cm. Den lille formfaktoren gjør at maskinvaren som kan plasseres i satellittene er begrenset. Dermed må all program og maskinvare som brukes på satellittene være tilpasset disse begrensningene.

Problemstillingen denne masteroppgaven tar for seg er korreksjon av feilregistrering og avvik som blir tatt opp av den hyperspektrale imageren som er på satellittene. Det er spesifikt korreksjonen av de såkalte smile og keystone effektene som blir fokusert på. I forkant av denne masteroppgaven ble det gjennomført en gjennomførbarhetsstudie, der en programvareløsning for korreksjonene ble utviklet. Dessverre ble korreksjonene ved hjelp av denne løsningen ikke utført raskt nok. Noe som ledet til denne masteroppgaven, som har som mål å implementere en løsning på programmerbar logikk.

Korreksjonene er delt inn i to hovedfunksjoner. Denne første funksjonen generer korreksjonsmatrisen, noe som blir gjort i programvare som ble utviklet i gjennomførbarhetsstudien. Denne delen vil dermed ikke være en del av arbeidet gjort i masteroppgaven. Den andre funksjonen gjør de faktiske korrigeringene. Korrigeringen er gjort ved å multiplisere korreksjonsmatrisen med en ukorrigert hyperspektralramme.

Korreksjonene på programmerbar logikk er gjort ved å sende data, som trengs for korreksjonene, fra systemminne til programmerbar logikk. Dette blir gjort ved hjelp av en AXI DMA. På den programmerbare logikken blir så selve korreksjonen utført og sendt tilbake til systemminne, via AXI DMA'en. Den endelige løsningen har kun en tråd, og er tregere enn programvareløsningen. Grunnen for dette er begrenset data overføringshastighet.

Preface

This master thesis is written as part of the HYPSO mission at NTNU, in the period 15.January to 01.July. The program of study, leading up to this master thesis, is the 2-year master in Electronic Systems Design, with chosen a specialisation in Design of Digital Systems.

I would like to thank all the members of the HYPSO project for all the support and help I have received throughout the past year. A lot about teamwork and debugging was learned.

I would also like to give special thanks to Milica Orlandic and Joseph Garrett for the counselling and advice I received during the feasibility project and the master thesis.

Contents

Abstract i

Sammendrag ii

Preface iii

List of Figures vi

List of Abbreviations vii

1 Introduction 1
1.1 Project Description 2

2 Theory and Background 3
2.1 Hyperspectral Imaging 3
2.1.1 Calibration 4
2.1.2 File Formats 4
2.2 Smile and Keystone 5
2.2.1 Detecting Smile and Keystone Effects 6
2.2.2 Correcting Keystone and Smile with Sparse Matrices 6
2.2.3 Benefit of using Sparse Matrix Multiplication 7
2.2.4 Sparse Matrix Formats 8
2.2.5 Compressed Sparse Row 8
2.2.6 Compressed Interleaved Sparse Row 9
2.3 Software Implementation 9
2.3.1 Improvements to the Software 9
2.4 Memory Access on Programmable Logic 10
2.4.1 Direct Memory Access 10
2.5 AXI Protocol 13
2.6 Hardware and Software used for Development 14
2.6.1 Vivado 14
2.6.2 ZedBoard 14
2.6.3 MATLAB 15
2.7 Goals and Requirements 16

3 Method 17
3.1 Hardware Implementation 17
3.1.1 Multiplication with Float Values 18
3.1.2 Correction Core 19
3.2 Software Driver 20
3.3 Verification 21
3.4 Alternative Attempted Solutions 22

4 Results 23
4.1 Performance and Resource Requirements of Software Implementation 23
4.2 Performance and Resource Requirements of Hardware Implementation 24

5 Discussion 26
5.1 Performance of Hardware 26
5.2 Future Work 27

5.3 Future Improvements 27
5.4 Alternative Implementations 28

6 Conclusion **29**

References **30**

Appendix **32**

Appendix A Testbench Correction Modules **A1**

Appendix B Register overview CubeDMA [1] **B1**

List of Figures

1.1 Illustration of HYPSO CubeSat. Taken from [2] 1
1.2 Illustration CubeSat concept of operations. Taken from [3] 1
1.3 Illustration of smile and keystone distortion. Taken from [4] 2

2.1 Illustration of respectively pushbroom and whiskbroom pointing. Taken from [5] 3
2.2 Illustration hyperspectral cube. Taken from [6] 3
2.3 Illustration of smile and keystone effect. Taken from [6] 5
2.4 Overview of CubeDMA. Taken from [7] 11
2.5 Illustration of block descriptors used for scatter-gather transfer. Taken from [7] . 12
2.6 Overview of AXI DMA. Taken from [8] 13
2.7 Waveform diagram of AXI handshake 14
2.8 The ZedBoard 15

3.1 Block diagram of complete system 18
3.2 Block diagram of Correction Core 19
3.3 Block diagram of multiplication module 20
3.4 Block diagram of test setup 21

4.1 Uncorrected hyperspectral cube 23
4.2 Corrected hyperspectral cube 23
4.3 Corrected hyperspectral frame, corrected in software 24
4.4 Corrected hyperspectral frame, corrected on programmable logic 24
4.5 Simulated results correction core 25

List of Abbreviations

HYPSO	HYPer-spectral Smallsat for ocean Observation
HSI	Hyper Spectral Imager
NTNU	The Norwegian University of Science and Technology
CSR	Compressed Sparse Row
CSC	Compressed Sparse Column
CISR	Compressed Interleaved Sparse Row
GSP	Geometric Control Points
FPS	Frames Per Second
MB	MegaBytes
GB	GigaByes
ms	MilliSeconds
DMA	Direct Memory Access
MM2S	Memory Map to Stream
S2MM	Stream to Memory Map
HDL	Hardware Description Language
SoC	System on Chip
OS	Operating System
PL	Programmable Logic
RAM	Random Access Memory
FPGA	Field-Programmable Gate Array
AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture

1. Introduction

This thesis is written for the HYPSO (HYPer-spectral SmallSat for ocean Observation) mission at NTNU. HYPSO is a mission managed by the NTNU SmallSat group, which consists of students, PhD's and professors who work on space and satellite related projects. The HYPSO mission itself has the goal of launching up several CubeSats. These CubeSats are built in accordance with the NASA CubeSat standard [9] and have a size of 6 U, where 1 U means a 10 cm cube with a weight of less than $1^{-1.33}$ Kg. The dimensions of the satellites will be 10 cm x 20 cm x 30 cm, Figure 1.1 shows an illustration of the CubeSat. The satellites have a Hyper Spectral Imager (HSI) which is meant to gather hyperspectral data from the ocean surface and send that data back to earth. Here, amongst other things, it will be used to record and predict algae bloom. This would make it possible to alert salmon farms of possible algae blooms coming their way. Predicting algae bloom would make it possible to move the salmon before the bloom hits the farms, which worst case, might end up killing all the salmon at a farm. See Figure 1.2 for an illustration of the CubeSats concept of operations.

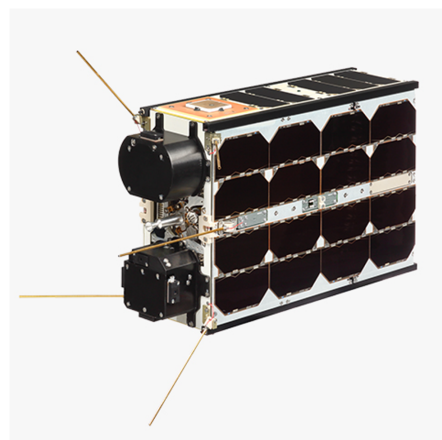


Figure 1.1: Illustration of HYPSO CubeSat. Taken from [2]

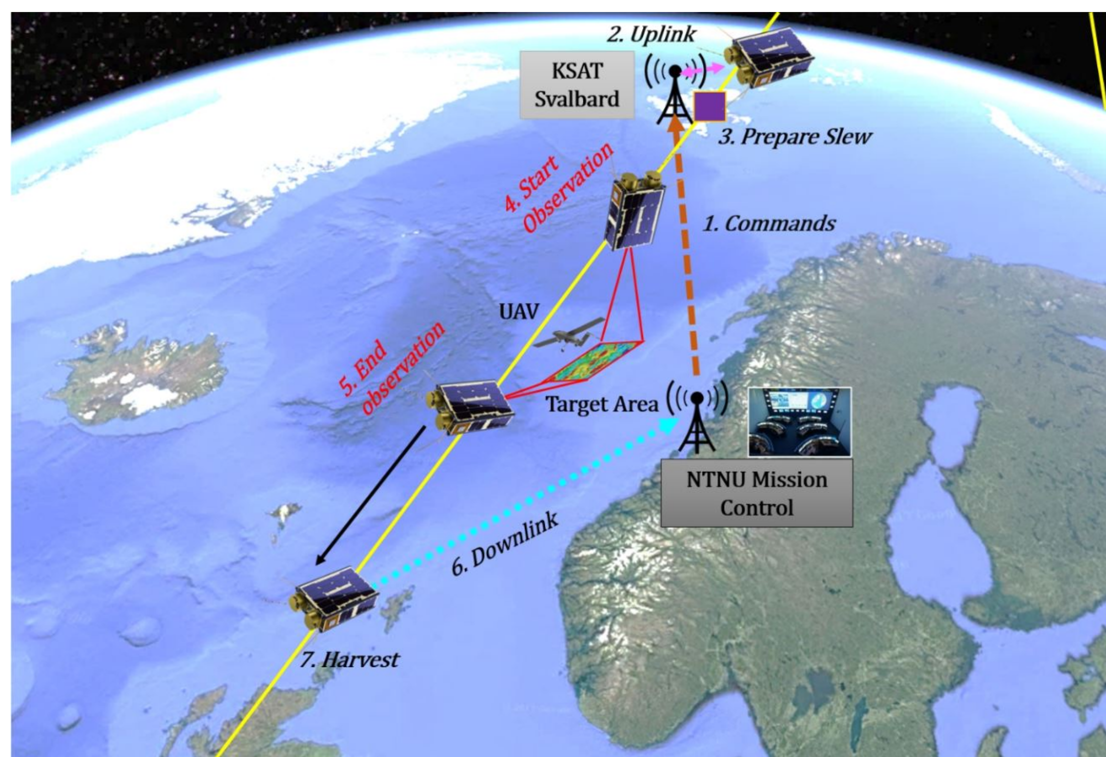


Figure 1.2: Illustration CubeSat concept of operations. Taken from [3]

The identification of algae blooms is done by detection and classification, based on the hyperspectral data that is collected. Algal blooms commonly produce specific displays of colour, due to the algae cells gathering on the water surface. Only a very small percentage of algae

produce harmful toxins, but due to the amount of algae found in algal blooms, these toxins can be harmful to fish and human consumers.

1.1 Project Description

One of the problems with hyperspectral imaging instruments is that they often suffer from spectral and spatial misregistrations. These may be caused by aberrations or misalignment of the optical hardware. This can in turn cause distortions in the spectral signatures of the target object, which means that the classification algorithm used to identify the object may fail.

A typical spectral misregistration is smile, which is the change in central wavelength in a spectral channel as a function of slit height [4]. A typical spatial misregistration is keystone, which is a change in position of the same spatial pixel in the scene as a function of wavelength. See Figure 1.3 for and illustration of smile and keystone effects.

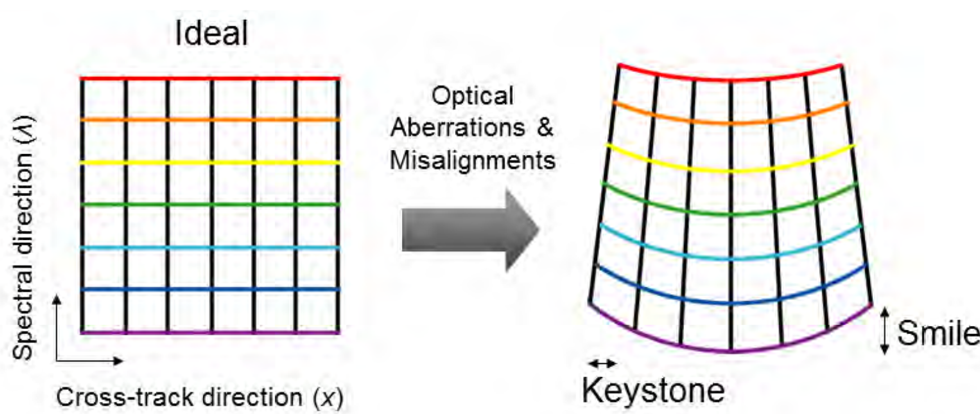


Figure 1.3: Illustration of smile and keystone distortion. Taken from [4]

This thesis is a continuation of the work done in a feasibility project preceding this master thesis [10]. The feasibility project focused on implementing corrections for smile and keystone effects in software. The software did not end up working correctly and as such, part of this thesis is to finish the software implementation. However, while the software implementation did produce incorrect results it was possible to obtain information on how long corrections take, and how much memory is required.

The main goal of the thesis is to create an FPGA implementation, which is faster than the software solution. Both the software and hardware solution have the goal of working on the ZedBoard, with a ZYNQ 7020 chip.

The correction consists of two parts. The first part creates a correction matrix which is used to correct the captured frames. The correction matrix does not need to be recalculated often, as it only changes when the hardware changes due to physical stimulus or time. As such this part will only exist in software, as the time requirements for it are less strict.

The second part is multiplying the correction matrix with the captured frame. Implementing this part on hardware is the main part of this thesis.

2. Theory and Background

This chapter presents the theory and background knowledge that the work of this thesis is based on. The sections up until section 2.3 cover theory found during the feasibility project [10] and work done in [6]. As such these sections will be similar to sections found in those reports. The sections following contain the theory that is specific to the FPGA implementation. Finally, the goals and requirements of the thesis will be presented.

2.1 Hyperspectral Imaging

Hyperspectral imaging is a technique that can be used to obtain spectral bands that cover the same spatial area at roughly the same time. The spectral information from these spectral bands can then be used by classification algorithms to detect and identify objects in said spectral area. For more information on this see [6], [11], [5], [12] and [13].

The spectral bands over the same spectral area at roughly the same time will hereafter be referred to as a hyperspectral frame, or just frame. By taking several such frames, a hyperspectral cube is obtained. These cubes can be obtained with various methods. Two such methods, pushbroom and whiskbroom pointing, are shown in Figure 2.1. For more information, refer to [6].

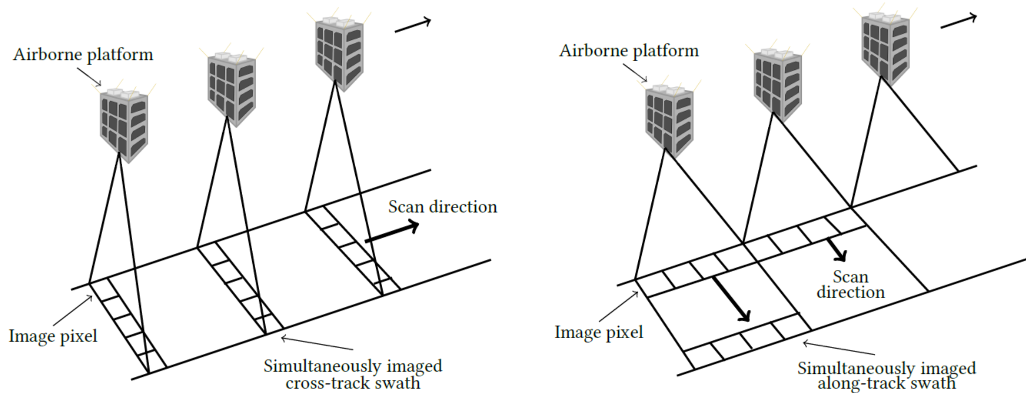


Figure 2.1: Illustration of respectively pushbroom and whiskbroom pointing. Taken from [5]

Pushbroom pointing is used when the HSI is based on a design with the slit mounted in cross-track with the moving direction. This means a complete frame is captured each time, with a complete cube being the result of a series of frames being taken. The HSI used in the CubeSat for this project is a pushbroom based design. According to [14], pushbroom is advantageous for earth observation purposes. Figure 2.2 shows an illustration of a hyperspectral cube. The spectral information is displayed on the x-axis, while the y-axis represents spatial information based on the slit height of the HSI. The z-axis shows the spatial information in the moving direction.

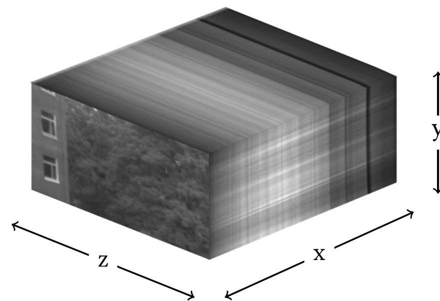


Figure 2.2: Illustration hyperspectral cube. Taken from [6]

2.1.1 Calibration

Hyperspectral imaging instruments often suffer from spectral and spatial misregistrations. These misregistrations can show up as aberrations and misalignments. To reduce the misregistrations, calibration is needed. Calibration is performed by converting measured data to physical and more meaningful units or information. This is done by comparing known measurements to the data obtained by the HSI. Calibration is important to make sure that the classification algorithms can obtain as accurate results as possible. The calibration of relevance for this thesis includes spectral and spatial calibration, which corrects the relationship between wavelength and pixel position. According to [5] a selection of light sources with known spectral peaks can be used to measure the response at selected wavelengths. While it is impossible to remove or correct all errors and misrepresentations, as much as possible should be characterised and corrected [4].

2.1.2 File Formats

The uncorrected and corrected spectrogram, as well as H_{ks} are saved in BIP (band-interleave-by-pixel) format. This is a binary file format that comes with a header file (.hdr). The header file contains information on how the BIP file is to be read. The information contained in the header file is as follows,

- Samples - number of columns
- Lines - number of rows
- Bands - number of bands
- Data type - the following are supported
  - 1 - 1-byte unsigned integer
  - 2 - 2-byte signed integer
  - 3 - 4-byte signed integer
  - 4 - 4-byte float
  - 5 - 8-byte double
  - 9 - 2x8-byte complex number made up from 2 doubles
  - 12 - 2-byte unsigned integer
- Header offset - number of bytes to skip before data starts in binary file
- Interleave - Permutations of dimensions in binary data
  - BSQ - Band Sequential (col, row, band)
  - BIL - Band interleave by line (col, band, row)
  - BIP - Band interleave by pixel (band, col, row)
- byte order - little or big-endian
  - 0 - little-endian byte order
  - 1 - big-endian byte order

BIP files read and write data in the order of bands, columns and then rows, this is the same way the HSI on the CubeSat stores its data. This means that data can be read sequentially and

that there is no need to search for specific bytes in the file. The only time searching has to be done is when a new frame is started. The configuration of the BIP file used in the CubeSat are 2-byte unsigned integer, no header offset, BIP, little endian, with the rows, columns and bands being unknown before the header file is checked.

2.2 Smile and Keystone

Smile and keystone distortions are two typical effects of spectral and spatial misregistration, Figure 2.3 shows an illustration of these effects. The coloured circles in the figure represent different wavelengths. The blue circles represent short wavelengths, while the green and red circles represent medium and long wavelengths respectively. An ideal system would result in a figure where all circles are perfect circles, placed perfectly in the middle of their grid square. This way all the dotted lines would be in parallel with the grid and completely straight.

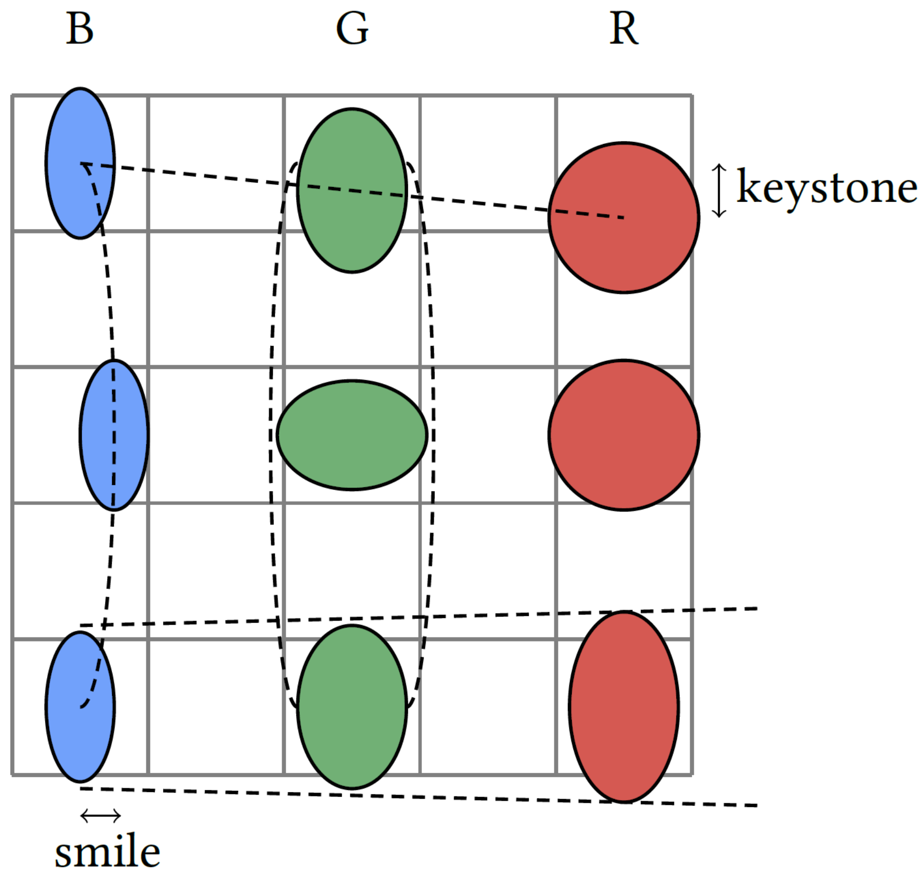


Figure 2.3: Illustration of smile and keystone effect. Taken from [6]

Smile is a typical misregistration in the spectral domain, caused by a change in the central wavelength. Visually this can be observed as a bent line in the spectrogram. Keystone is a typical misregistration of the spatial domain, and can be observed as a slight misplacement of objects. Keystone effects can be caused by difference in slit magnification of wavelengths. [15]. Typically, the tolerance of these errors is a small fraction of a pixel, less than 2% of a pixel for smile effects and less than 5% of a pixel for keystone effects [16].

2.2.1 Detecting Smile and Keystone Effects

For the work done in this thesis smile and keystone effects are obtained simultaneously, the method for this is based on work done in [6], as such more detailed information on this can be found there.

The detection is done by finding known control points in the image. These points are known as Geometric Control Points (GCPs). Using these points it is possible to determine the smile and keystone effects for a given HSI. The smile and keystone effects change slowly over time, as such one can use the same corrections over a longer period of time, as long as the imager used stays the same. This method is described in [11]. The distortion in the original image determined by the GCPs can be described by the following 2-D quadratic polynomial distortion model.

$$x = a_{00} + a_{10}x_{ref} + a_{01}y_{ref} + a_{11}x_{ref}y_{ref} + a_{20}x_{ref}^2 + a_{02}y_{ref}^2 \quad (2.1)$$

$$y = b_{00} + b_{10}x_{ref} + b_{01}y_{ref} + b_{11}x_{ref}y_{ref} + b_{20}x_{ref}^2 + b_{02}y_{ref}^2 \quad (2.2)$$

Here x and y are the measured coordinates, while x_{ref} and y_{ref} are the known reference coordinates, a and b are the model coefficients.

In matrix form this can be written as in Equation 2.3, Where W is an NxM matrix, N being the polynomial degree and M being the number of GCPs. X and Y are matrices are the same size as the image frame, and they hold the indices of the pixels of the image frame. Applying error terms and solving for the pseudo-inverse solution then comes out as Equation 2.5.

$$X = WA \quad (2.3)$$

$$Y = WB \quad (2.4)$$

$$A_{hat} = (W^T W)^{-1} W^T X \quad (2.5)$$

$$B_{hat} = (W^T W)^{-1} W^T Y \quad (2.6)$$

A_{hat} and B_{hat} can then after they are found, be used to apply corrections.

2.2.2 Correcting Keystone and Smile with Sparse Matrices

To correct the spectrogram, matrix multiplication is used. The raw spectrogram is written as a 1-D vector, U, with N spectral bands and M spatial pixels, for a total of NM pixels. The matrix H_{ks} which will contain the smile and keystone corrections is generated. This matrix has to be generated once initially, and thereafter only if drift in the hardware causes the smile and keystone effects to change significantly, or if the frame size changes. As many elements in H_{ks} are zero and the total size of H_{ks} is in the order of 10^{12} elements, sparse matrices are used to ease computation and storage. The corrected spectrogram V is obtained by multiplying H_{ks} with the vector U, V= H_{ks}U.

A second-order polynomial can be calculated in order to generate a pixel-to-pixel map from the uncorrected spectrogram to the corrected spectrogram. An approximated two-dimensional quadratic polynomial distortion model is given by Equation 2.7.

$$f_\eta(p_\lambda, p_y) = \sum_{l,k=0}^{l+k\leq 2} a_{\eta lk} p_\lambda^l p_y^k \tag{2.7}$$

Here p_λ and p_y are the pixel indices of the uncorrected spectrogram. η indicates either λ or y. f_λ and f_y identify the position of the pixel in the corrected spectrogram. Finally, the $a_{\eta lk}$ values are the calibration coefficients A_{hat} and B_{hat}.

H_{ks} is calculated from this polynomial map. Since the pixels in the uncorrected spectrogram are mapped to position that may be in between several pixels, the calculated correction is split between four pixels in the corrected spectrogram. Four pixels are used to fully correct spatial distortion and introduce minimal blur [17]. The four pixel correction values and positions are calculated as follows.

$$\begin{aligned} H(r_\lambda, r_y)j &= (1-\Delta_\lambda)(1-\Delta_y) \\ H(r_\lambda+1, r_y)j &= \Delta_\lambda(1-\Delta_y) \\ H(r_\lambda, r_y+1)j &= (1-\Delta_\lambda)\Delta_y \\ H(r_\lambda+1, r_y+1)j &= \Delta_\lambda\Delta_y \end{aligned} \tag{2.8}$$

Where

$$r_\eta(p_\lambda, p_y) = floor(f_\eta(p_\lambda, p_y)) \tag{2.9}$$

$$\Delta_\eta(p_\lambda, p_y) = f_\eta(p_\lambda, p_y) - r_\eta(p_\lambda, p_y) \tag{2.10}$$

The total correction value of the four pixels is always 1, meaning the total radiance is preserved. The values of H_{ks} that do not obtain a value from the polynomial map are set to 0. As such the resulting matrix, with the size of $(rows(y) * bands(\lambda))^2$, will have a majority of elements that are zero. The rows in H_{ks} correspond to the pixels in the uncorrected spectrogram vector (U) with the same index. Each row in H_{ks} will only ever have four non-zero elements. These four elements may however be mapped to different pixels in the corrected spectrogram.

2.2.3 Benefit of using Sparse Matrix Multiplication

A matrix is considered sparse when the majority of elements in it are zero. If most elements are nonzero the matrix is considered dense. Sparse matrix multiplication refers to the act of only multiplying the nonzero elements, as multiplying with zero always returns zero. If a result is known before calculating it, there is no point in doing the calculation. As an example take the 5×5 matrix M in Equation 2.11.

$$M = \begin{pmatrix} 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 5 & 0 & 3 & 2 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 & 2 \end{pmatrix} \tag{2.11}$$

Here there are 10 nonzero values and 15 zeros, making it a sparse matrix. If we multiply the matrix with a 1-D vector V=(1 3 5 3 2) we would need to do 25 multiplication operations. However, if sparse matrix multiplication is used the number of multiplication operations goes down to 10. The example matrix given is very small, in reality matrices often contain several

million elements, of which often less than 1% are nonzero. Not computing the nonzero elements can therefor result in very significant speedups.

2.2.4 Sparse Matrix Formats

Since H_{ks} can easily reach a size of 10^{12} elements, where a majority of elements are zero, sparse matrix ordering and operations are used for the correction process. There are several methods to order and operate on spare matrices, the once of relevance for this thesis are the triplet format and CISR, the information here is largely based on [18] and [19].

Triplet Format

The triplet format, also called coordinate list or COO for short, stores the row, column and value of all non-zero elements in three separate 1-D arrays. This means, if we have matrix A in Equation 2.12, we end up with the three 1-D arrays in Equation 2.13.

$$A = \begin{pmatrix} 4 & 0 & 5 & 0 \\ 0 & 0 & 7 & 0 \\ 3 & 2 & 0 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix} \tag{2.12}$$

$$\begin{aligned} row &= \begin{pmatrix} 0 & 0 & 1 & 2 & 2 & 3 \end{pmatrix} \\ col &= \begin{pmatrix} 0 & 2 & 2 & 0 & 1 & 1 \end{pmatrix} \\ val &= \begin{pmatrix} 4 & 5 & 7 & 3 & 2 & 6 \end{pmatrix} \end{aligned} \tag{2.13}$$

The dimensions of the original matrix need to be stored in order to be able to reconstruct the original matrix. The reason for this is that there might not be a non-zero element in the first or last row or column. This format is easy to construct and saves all the indices of the non-zero values, which makes it easy to reconstruct the original matrix, as long as the dimension of said matrix are available. For matrix multiplication it removes the need to multiply elements with zero, saving computing time. More memory is required for this format, compared to other formats, as both row and column indices for each non-zero values are stored. For a single threaded application the difference in speed compared to other methods is rather small, but it will still be on the slower end. Triplet ordering does fall behind for multi-threaded applications.

2.2.5 Compressed Sparse Row

Compressed Sparse Row (CSR) stores the data in three 1-D arrays. Column and value arrays are still determined in the same way as in the triplet format, the row array contains an index for the column and value arrays indicating the starting point of each new row of values. Using matrix A in Equation 2.12 as an example, we get the three arrays in Equation 2.14.

$$\begin{aligned} row &= \begin{pmatrix} 0 & 2 & 3 & 5 \end{pmatrix} \\ col &= \begin{pmatrix} 0 & 2 & 2 & 0 & 1 & 1 \end{pmatrix} \\ val &= \begin{pmatrix} 4 & 5 & 7 & 3 & 2 & 6 \end{pmatrix} \end{aligned} \tag{2.14}$$

Since only the starting points of new rows have to be stored, rather than indices for each element, a significant amount of memory can be saved in the case of larger matrices. The main challenge with CSR is the construction, which will be more complex and time-consuming than constructing a COO format matrix.

2.2.6 Compressed Interleaved Sparse Row

Compressed Interleaved Sparse Row (CISR) [20], is an evolution of CSR. CISR also orders data into three 1-D arrays. The row array contains how many non-zero elements are in the given row of the original matrix, while the column array contains the column index and value contains the value of the non-zero element. Using matrix A in Equation 2.12 as an example, we get the three arrays in Equation 2.15.

$$\begin{aligned} row &= \begin{pmatrix} 2 & 1 & 2 & 1 \end{pmatrix} \\ col &= \begin{pmatrix} 0 & 2 & 2 & 0 & 1 & 1 \end{pmatrix} \\ val &= \begin{pmatrix} 4 & 5 & 7 & 3 & 2 & 6 \end{pmatrix} \end{aligned} \tag{2.15}$$

The main benefit of CISR is that it makes parallelization easier. Since the number of non-zero elements in a row are known, several rows can easily be operated on simultaneously. Knowing the amount of non-zero elements per row makes it easy to spawn the required amount of threads, which makes this format very useful and fast for multi-threaded applications. For single threaded applications there is little benefit in using this format.

Compressed Sparse Column (CSC) was also explored [10], but was decided against due to CISR having the highest gain in speed when multi-threading the sparse matrices in the feasibility project.

2.3 Software Implementation

During the feasibility project preceding this master thesis a software solution was developed [10], to determine if there is a need for a solution on programmable logic. The proposed implementation uses the aforementioned triplet format and is single threaded. This implementation did end up not working correctly, as the hyperspectral frames resulting from the correction were unrecognisable. However, the achieved functionality is still sufficient to draw a few conclusions. The time to correct a single frame on the target hardware is estimated to be around 700 ms, while the desired speed is 65 ms per frame, or less. Even with multithreading it is unlikely that the software implementation can reach the desired speed.

Further it was also determined that operation on a complete frame with the dimensions 1088×2048 (rows, bands) requires roughly 100 MB of RAM. Since the target hardware has 500 MB available both software and hardware solution should work equally well as far as memory is concerned.

2.3.1 Improvements to the Software

Before a solution on programmable logic could be attempted, the software solution had to be fixed. This was necessary to ensure that the method of implementation was correct, to avoid making the same mistake on programmable logic.

The problem turned out to be that the BIP file used for testing defined which direction column and rows are different than the BIP files that are generated on the satellite. As rows and columns are the spatial dimensions, it is up to the user to define which of the y and z-axis correspond to rows and columns. For the satellite the z-axis corresponds to rows and the y-axis corresponds to columns Figure 2.2, the test BIP file defined it the other way around.

Once this issue had been determined, a BIP file with the correct order was used to test the software implementation, which now produced the correct result.

Additionally, the following changes and improvements were done:

- A system generated header file, which contains the cube dimensions, is now used. This means the program does not have to be rebuild every time the dimensions change. It is enough to update the header file.
- The data of the uncorrected frame is now binned due to hardware restrictions [21], this does not change the functionality of the program, but could make the corrections less accurate. This was not tested in depth due to time restrictions.
- The correction matrix can now be saved in CISR format, in addition to triplet format.
- The correction matrix saved in CISR format has its values multiplied by 2^{14} and type changed from float to integer, to make multiplication on programmable logic simpler.

Finally, it was discovered during work on programmable logic, that transferring and storing the corrected samples repeatedly was not a feasible solution. As such a new solution was needed.

By reordering the correction matrix, so that each sample of the corrected matrix is corrected in one go, the need to store the corrected frame in RAM disappears, and the corrected value can be written directly to file on the SD-card. This however means that it is not possible to use CISR format. The reason for that is that in the correction matrix the row index corresponds to the index of the sample in the uncorrected frame, the column index corresponds to the index of the sample in the corrected frame and the value found at the position is the correction value. Since all pixels in the uncorrected frame are used for correction, it is possible to use CISR format. Because a counter can take care of determining which uncorrected sample is needed next. For the corrected frame on the other hand, there are some samples that are not corrected, meaning that it would be impossible to determine which sample is being corrected, if the CISR format is used. As such the reordered correction matrix needs to be in triplet format. Alternatively, it would be possible to include the samples of the corrected frame, that are not corrected, in the CISR format matrix. Meaning there would be elements with the value zero, making it no longer a sparse matrix or true CISR format. The benefits of CISR would also disappear by doing this, as more memory space and decoding time would be needed, compared to the triplet format.

The software does the reordering by simply gathering the corrections for each sample of the corrected frame, from the original triplet matrix, and ordering them sequentially. See Equation 2.16 for an example, corr being the array with the index of the corrected frame, uncorr being the index of the corrected frame and val being the correction value.

$$\begin{aligned} corr &= \begin{pmatrix} 1 & 1 & 3 & 4 & 4 & 4 \end{pmatrix} \\ uncorr &= \begin{pmatrix} 0 & 2 & 2 & 0 & 1 & 1 \end{pmatrix} \\ val &= \begin{pmatrix} 4 & 5 & 7 & 3 & 2 & 6 \end{pmatrix} \end{aligned} \tag{2.16}$$

2.4 Memory Access on Programmable Logic

Programmable logic does not automatically have access to system memory. To be able to store data the programmable memory has to be given access to memory, or memory has to be developed on the programmable logic. For this thesis, memory access is done via Direct Memory Access (DMA).

2.4.1 Direct Memory Access

In general, there are three main ways of obtaining data for I/O devices, polling, interrupts and DMA. Polling dedicates the processor of the system to acquire data, often waiting in a loop. Interrupts stop the processors current task when new data is required, meaning the processor can do other tasks between fetching data requests. For DMA a dedicated data transfer device handles incoming and outgoing data and stores it in system buffers until the processor retrieves

it.

For high speed applications or software/hardware co-designs it is highly beneficial to have a DMA on-board, as the DMA is a separate dedicated piece of hardware, which handles data transfers. Since the DMA is separate from the processor, this also means that the processor does not need to execute instructions for data transfer, freeing up time for other tasks.

There are several types and implementations of DMA, with different use cases. For this project there are mainly two implementations that are looked at. Firstly, the CubeDMA, which was developed specifically for the use on the HYPSO-SmallSat. Secondly the AXI DMA which is and existing and well tested IP by Xilinx, that is provided with Vivado. Both of these implementations should have the required functions needed for this project.

CubeDMA

The CubeDMA was developed by a previous master student working on the HYPSO-project [7]. Figure 2.4 shows an overview of the CubeDMA. This DMA has two independent channels, MM2S (Memory Map to Stream) and S2MM (Stream to Memory Map). These channels do what their names suggest. MM2S takes data from memory and streams it to a processor or accelerator. S2MM streams data from the processor or accelerator to memory.

Both channels can handle component sizes that are not byte multiples. The MM2S channel can handle BIP and BSQ ordered transfers. For the BSQ ordered mode it is possible to stream several planes, or bands, in parallel. This means that several bands of the same pixel are sent out in parallel. For the BIP mode this is also possible. The difference being that in BIP mode, when doing parallel transfers, the bands of only one pixel are transferred, while bands of several pixels can be transferred in BSQ modes. The amount of data transferred per cycle is the same however.

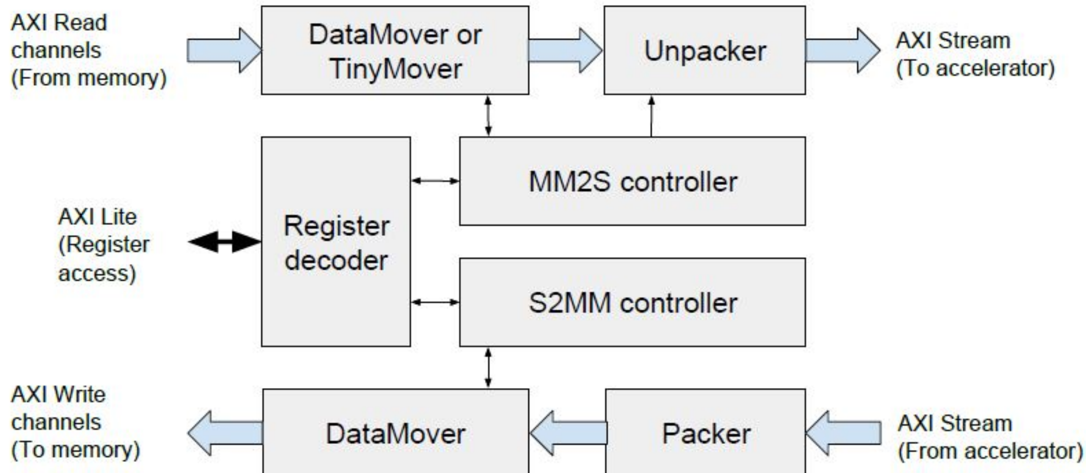


Figure 2.4: Overview of CubeDMA. Taken from [7]

For both modes data can be ordered sequentially or block-wise, and it is possible to define an initial offset for both modes. Sequential mode will start at the first pixel, upper left corner, and proceed until the last pixel is reached, lower right corner. For block-wise transfer the cube is divided into blocks, with bounds in the x- and y-direction. The whole of the z-direction being taken regardless. Each individual block is transferred sequentially. Block dimensions have to be powers of two.

In parallel with the data stream from the DMA, a control stream is sent, with control bits that indicate whether the component is the last pixel of a block. This makes it possible to handle

data that is smaller than a given block size.

The S2MM channel writes data sequentially, supports data words of different sizes and collects data into 64 bit packets, that are then stored in memory. For a registry overview see Appendix B. For a more thorough overview of how the CubeDMA is implemented see [7].

Table 2.1 shows how AXI DMA and CubeDMA performance stack up against each other. As can be seen, for sequential BIP transfers there is no difference in speed. The difference in speed when blocking the cube is not relevant to this project, as this functionality will not be used. The same goes for the fact that the AXI DMA has no BSQ mode, as all data will be read and written in BIP format.

Table 2.1: Performance of CubeDMA and AXI DMA

	BIP seq	BIP block	BSQ seq
Theoretical	800 MB/s	800 MB/s	100 MB/s
AXI DMA	800 MB/s	340 MB/s	N/A
CubeDMA	800 MB/s	775 MB/s	14.1 MB/s

AXI DMA

AXI DMA is a DMA solution from Xilinx and unlike the CubeDMA it has the option of scatter-gather transfers, which means that data can be read or written non-sequentially. This is done by using block descriptors, which describe the start point, the length of the block and the next segment, which determines if another block should be started. Several such block descriptors can be chained together to form a continuous stream, without the need to look up and transfer each block separately, see Figure 2.5. The drawback of using block descriptors is that CPU intervention is required to set up the block descriptors every time a new one is needed. As such block transfers and scatter-gather are significantly slower than sequential streaming, as previously seen in Table 2.1.

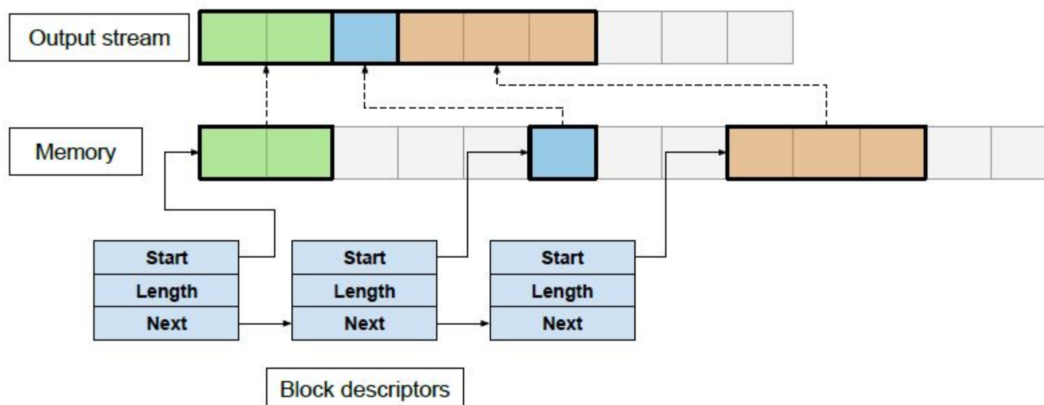


Figure 2.5: Illustration of block descriptors used for scatter-gather transfer. Taken from [7]

An overview of how the AXI DMA works is shown in Figure 2.6. As is the case with the CubeDMA, the AXI DMA also has two independent channels, MM2S and S2MM. Together these two channels can support up to 16 paths in scatter-gather mode, meaning 16 transfers can be done in parallel. The full function and register overview of the AXI DMA can be found in [8].

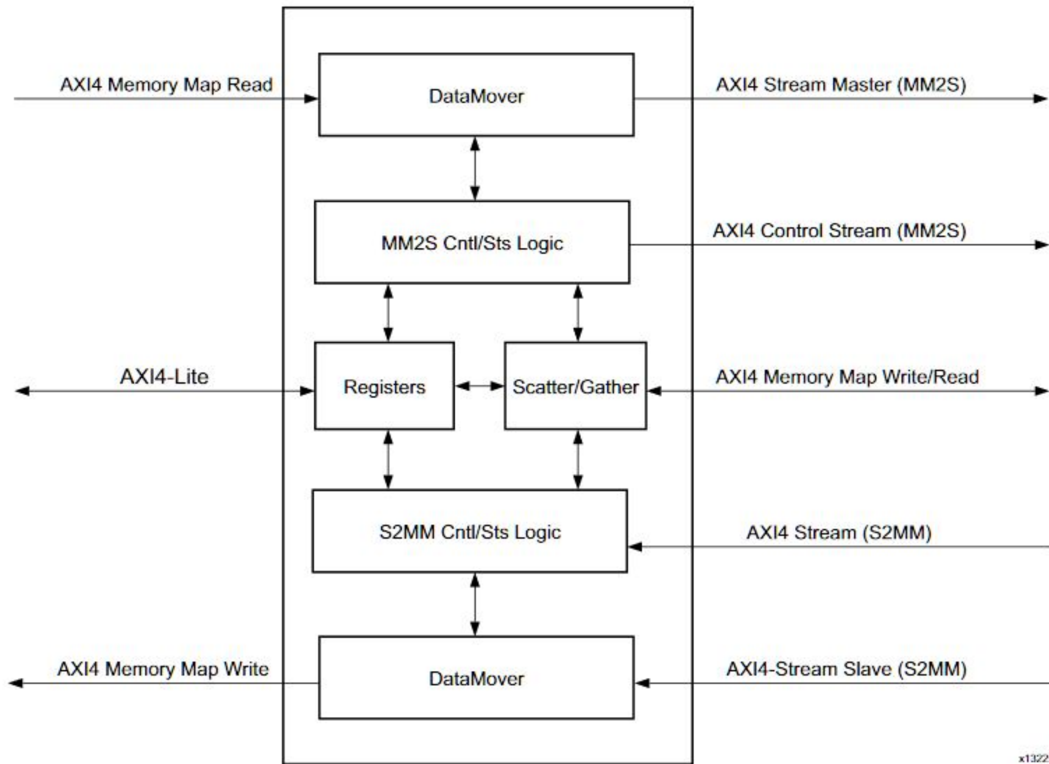


Figure 2.6: Overview of AXI DMA. Taken from [8]

Other Considered DMAs

There are several other DMA IPs that are provided by Xilinx, among these are the AXI Central DMA (CDMA), AXI Video DMA (VDMA) and the AXI Multichannel DMA (MCDMA). These DMAs all function similarly to the AXI DMA, but have a few key differences.

The reason that the CDMA was not chosen is that it has memory mapped interfaces on both sides, while that AXI DMA has a stream interface on one side and a memory mapped interface on the other. The stream interface on the AXI DMA simplifies the transfer of data for this use case.

The VDMA lacks scatter gather support, which is needed, due to the fact that the data needed for each correction is scattered, rather than gathered in one place.

The AXI MCDMA has all the required functionality and would allow for more data to be sent, but it is also more complex and requires more area on the FPGA. Also, the AXI DMA is able to transfer data fast enough for this use case. As such it was determined that the AXI DMA is the preferred choice.

2.5 AXI Protocol

The Advanced eXtensible Interface (AXI) is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) 4 specifications [22]. The AXi protocol is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface. Its main purpose is on-chip communication.

Axi has a wide variety of features, among these are the following.

- Separate address/control and data phases.
- Support for unaligned data.
- Burst-based transfers.
- Separate and independent read and write channels.
- Out-of-order transactions

The AXI protocol makes use of a so-called handshake to initiate and complete data transfer between master and slave. This handshake uses only two signals for control, the ready and valid signals. Only when both are HIGH, meaning they have the logical value 1, the data on the bus is transferred. The valid signal indicates whether the data present on the bus is valid, the ready signal indicates whether the receiver is ready to accept new data. Figure 2.7 shows a waveform of the handshake.

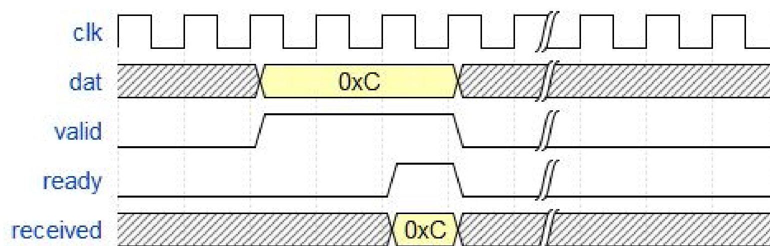


Figure 2.7: Waveform diagram of AXI handshake

2.6 Hardware and Software used for Development

This section contains a brief overview of the hardware and software used in the project.

2.6.1 Vivado

Vivado is a software suite produced by Xilinx. It is used for development, synthesis and analysis of HDL designs. In addition to basic tools needed for development of HDL designs it also contains tutorials, simulation tools, free IPs to use in projects, automation of connection between different IPs and a Software Development Kit (SDK), which allows software to be developed and run in parallel with hardware.

Vivado 2019.1 is the version that is used during this project.

2.6.2 ZedBoard

The development board used to test and run the design developed for this thesis is the ZedBoard Figure 2.8. The ZedBoard is a low-cost development board for the Xilinx Zynq-7000 programmable System on Chip (SoC). Applications for Linux, Windows and a number of other operating systems (OS) can be run on the ZedBoard. For testing purposes it is also possible to run applications bare-metal, that is to say without any OS, these can also called stand-alone applications. Running applications bare metal simplifies the testing process, as errors will be limited to the design only, but a design running on bare metal is not guaranteed to run on a given OS.

The ZedBoard features the following specifications:

- Zynq-7000 Programmable SoC XC7Z020-CLG484-1

- 512 MB DDR3
- 256 Mb Quad-SPI Flash
- 4 GB SD card
- USB-JTAG Programming
- 10/100/1000 Ethernet
- USB 2.0 and USB-UART
- PS & PL I/O expansion
- Multiple Displays
- I2S Audio Codec

The main features used are the programmable SoC, DDR3 memory, SD card, PL & PS I/O and the USB UART. All development and testing is done on bare metal.

Figure 2.8: The ZedBoard

2.6.3 MATLAB

MATLAB is a numerical computing environment and programming language that allows for matrix manipulation, plotting of functions and data, scripting and more. MATLAB allows for quick and easy analyses and illustration of data. Matlab is developed by MathWorks.

2.7 Goals and Requirements

The purpose of the thesis is to implement smile and keystone corrections on the programmable logic of the ZedBoard. The base for the corrections was developed during a feasibility project during the fall before this master thesis. The results from that project are used as a benchmark on how well the hardware implementation works, and if it is worth using over a software solution. The hardware solution developed needs to run on the target hardware of the HYPSO SmallSat, but does not need to take other applications into consideration, as different hardware solutions can be stored and loaded onto the hardware during flight [23].

The on-board processing is similar to the ZedBoard [24], with the main difference being that a ZYNQ 7030 [25] chip is used instead of a ZYNQ 7020 chip. This means more processing power and logic is available. For simplicity the ZedBoard is used as the testing device under development, as the ZYNQ 7030 should have no problems running the code if the ZYNQ 7020 can run it.

The main pieces of hardware that might limit the design are the processor and the memory. The processor is a Dual-Core ARMCortex-A9 MPCoreUp with speeds up to 1 GHz, which does not share the x86 instruction set of most computers that run on AMD or Intel chips. The memory is 1 GB of DDR3 RAM. The ZedBoard has 512 MB of DDR3 RAM, which means that as long as the program can run on the ZedBoard it can run on the on-board hardware.

Outside of hardware limitations the corrections should also run at more than 15 frames per second (FPS). This is not a hard limit, but rather a desirable goal. 15 FPS means that the correction per frame can only take 65 ms. This goal was not achieved in C code for the given processor. However, due to the ability to easily run parallel processes on an FPGA, it is likely that higher performance can be achieved.

In short,this report aims to answer the following:

1. Can an FPGA solution that corrects smile and keystone effects be implemented on the on-board hardware?

2. How many parallel operations can be done given the limitations of area and data transfer speed?

3. Is the FPGA solution fast enough to be worth using over the less complex software solution?

4. Is the AXI DMA or the CubeDMA better suited for this specific implementation?

Secondary goals and requirements are as follows.

1. Clean up and test software implementation on ZedBoard.

2. Self-checking for errors during computation.

3. Integrate FPGA solution into complete system.

These are not considered a main focus during the project and will only be considered when time allows for it.

3. Method

This chapter is divided into three main parts, the first one describes how the FPGA is programmed. Thereafter, the software driver that is needed to use the AXI DMA is presented and finally the methods of verification are presented.

3.1 Hardware Implementation

The first step of the hardware implementation is to determine how to access memory. The two main alternatives that were considered are the CubeDMA and the AXI DMA. They both function similarly when reading BIP files, which is the case for this thesis. When reading sequentially the performance and method of obtaining data is the same for both DMAs. The difference comes when data is read non-sequentially. The CubeDMA can read blocks of the complete cube, which is not possible to do directly in the AXI DMA. While the AXI DMA is unable to read block of the cube, it has the ability to use scatter gather mode, which makes it possible to pick out specific parts of the whole cube. The big difference in blocking and scatter gather is in how data is obtained, scatter gather needs block descriptors for each data part, as mentioned in subsubsection 2.4.1.

For this thesis two files need to be fetched for calculation, the first one is the correction matrix, this will always be read sequentially and in BIP format, as such there is no difference between the two DMAs. The second file is the uncorrected frame. This file will also be in BIP format, but the data needed from the uncorrected frame is not sequential. The correction is done one sample at a time for the corrected frame. However, each sample in the corrected frame is affected by several samples in the uncorrected frame, and the samples that are affecting the corrected sample may not be sequential.

There are two ways that are considered to obtain data, the first is to read the uncorrected frame sequentially and discard the data that is not required for the current correction. Doing it this way, both DMAs perform the same. The second way would be to target the specific data needed. This is something only the AXI DMA can do, with scatter gather.

The simplest solution would be a software/hardware co-design. Since a software driver is needed to run the DMA regardless, this driver may as well have the capability to generate the block descriptors needed for scatter-gather transfers without input from the hardware. This is done by letting the driver read the correction matrix and determining which data from the uncorrected frame needs to be sent. More on this in section 3.2. For this method to work, scatter-gather transfers are needed, as such the AXI DMA will be used for the implementation.

Another benefit of using the driver to determine which data is needed for correction is that the correction matrix does not have to be sent to the programmable logic. This means less time spent transferring data, and it simplifies the logic that needs to be generated.

A block diagram of the complete system can be seen in Figure 3.1.

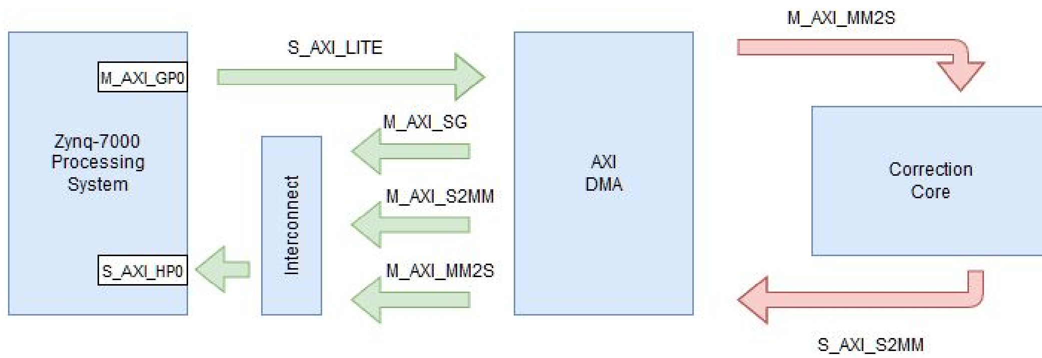


Figure 3.1: Block diagram of complete system

The processing system contains the ZYNQ 7020 chip set and all required software and files to keep the system. The GP0 pin (general purpose pin 0) is used as the hardware interface between the software and the programmable logic, connected to the DMA. GP0 is used to fetch data from memory and send it over the MM2S channel to the correction core. The correction core then sends the corrected data back to the DMA via the S2MM channel, the DMA then uses an AXI-interconnect to send data back to memory via the HP0 pin (high performance pin 0).

3.1.1 Multiplication with Float Values

As handling floats on programmable logic is significantly more difficult than handling integers, the easiest way to handle the float values from the correction matrix is to multiply them by a known value in the software. Allowing the PL to read the values as integers.

First it is necessary to determine how many decimals are needed to get good enough precision to not affect the corrections. The software implementation uses floats, meaning seven decimal precision is used, this might however be more than is needed. The important part is that the incident intensity is preserved, meaning that the total correction value in a row of the original correction matrix is always one [6]. A MATLAB script is used to check the maximum error in results from one to seven decimals. As can be seen in Table 3.1 using a four decimal precision, we get a maximum error of 0.03%. Considering the fact that smile effects need to be under 2% for the pixels, and that the polynomial used is not perfect to begin with, the error introduced by using only four decimal precision is acceptable. Since all correction values are less than one, representing the values with 16 bits is enough, as no value will exceed 65 535, when multiplied by 10 000.

Table 3.1: Error in corrections for different precision

Precision	1	2	3	4	5	6	7
Incident Intensity	1.30000	1.03000	1.00300	1.00030	1.00003	1.00000	1.00000
Difference in %	30	3	0.3	0.03	0.003	0	0

For programmable logic, multiplication is rather fast, regardless of the values multiplied, but to obtain the correct final values, the multiplied values from the correction matrix have to be divided by the same number at some point. Division can be a rather slow and complicated process on programmable logic. One solution would be to do the division in software, but the easier and more efficient way is to multiply and divide the correction values by a multiple of two. When multiplying or dividing binary values with powers of two, one can simply left shift the value n times in case of multiplication, and right shift n times in the case of division. Since four decimals are needed, 2^{14} (16 384) is used for multiplication and division, this still allows for the values to be represented in 16 bits.

3.1.2 Correction Core

The data is ordered in such a way that one sample of the corrected frame is completed at a time. This way there is no need to store and fetch samples of the corrected frame repeatedly. This is unlike the software implementation, where the corrections for one sample in the uncorrected frame is done at a time. The main reason for this is that no feasible solution was found to read and write the corrected frame repeatedly, in a timely manner. The method of obtaining the correction matrix is the same as for the software implementation, the only difference is that the data gets reordered, so that all corrections for each pixel in the corrected frame are grouped together. The drawback of this method is that it is unknown how many multiplications are needed for each corrected pixel, in the case of samples that are placed on the edge of a frame, as many as 1000 multiplications might be needed. However, seeing as the values for these samples are unimportant and likely wrong regardless, multiplication for these are skipped. This does not change the fact that the other corrected pixels are corrected an unknown amount of times.

By using a MATLAB script that checked how often non-edge samples are corrected, it was determined that 42 corrections was the maximum for the dimensions of the test frame. As a safety measure for larger cube sizes it was determined that up to 50 multiplications would be possible. Since the correction core needs both the correction value and the uncorrected data, a register with the size of 1 600 bits stores the incoming data. Since all multiplication is done in parallel adding a safety net does not affect the performance here, the same goes for the addition. More area is used by doing all multiplications and addition in parallel, however the increase was small enough to not result in problems with the area available. The increased area does nevertheless limit the amount of additional threads that could be implemented in the future.

The actual core consists of three modules. The first module buffers the data sent by the AXI DMA. A block diagram for the core can be seen in Figure 3.2. The block diagram shows only four multiplications, as showing all 50 would results in clutter and add no extra information.

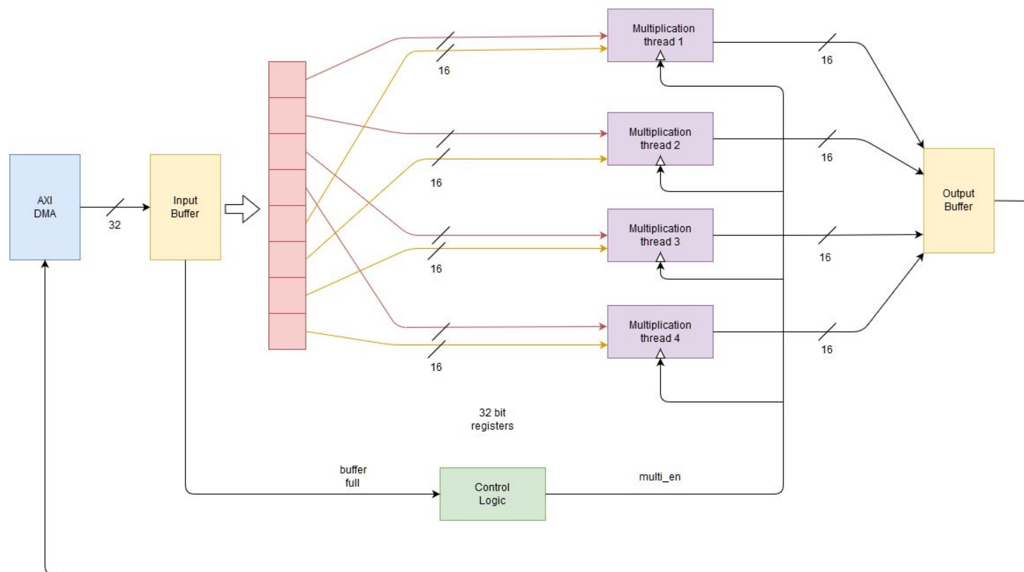


Figure 3.2: Block diagram of Correction Core

The DMA sends the data that is sent to the correction core to a buffer with the aforementioned size of 1600 bits. Once this buffer is full, the multiplication threads get the go ahead, each thread is bound to two 16 bit addresses of the buffer. The two 16 bit values are then multiplied and then sent to an adder. This adder then adds all the multiplied values together and right shifts

the result 14 times. The lowest 16 bits of the result are then sent to the output buffer, all other values than the least significant 16 bits should be zero. See Figure 3.3 for a block diagram of a multiplication module, again only four instances are shown, for simplicity's sake.

Once the completed samples have been obtained they are sent back to memory and stored in the file for the corrected cube.

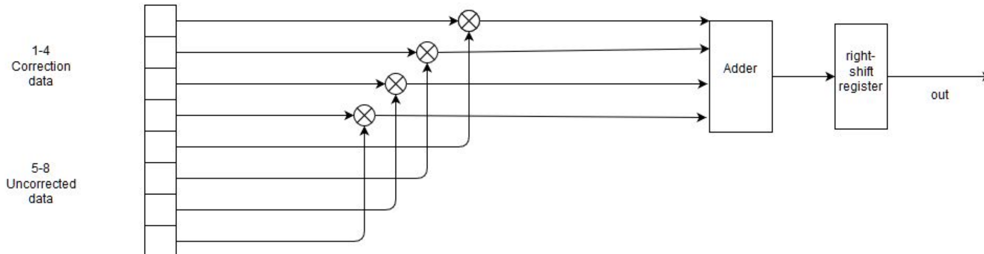


Figure 3.3: Block diagram of multiplication module

3.2 Software Driver

The AXI DMA needs a software driver to function. The driver used in this thesis is a modified version of the AXI DMA scatter-gather driver provided in Vivado. This driver contains the setup and initialisation of the DMA, as well as the setup of the receive channel (RX) and the transmit channel (TX). The aforementioned functions are used as are and no modifications are done.

The setup of the AXI DMA ensures that the hardware is correctly configured to scatter-gather mode, that the base address is correctly set and that the TX and RX are correctly configured.

The TX and RX setups both function the same for their respective channels. Base and High addresses are set for both channels, to ensure that data is not overwritten by accident. Interrupt or polling mode is set. For interrupt mode, transfers only happen when interrupts are sent out, which frees up the CPU for other tasks. In polling mode the CPU check whether the DMA is done with the transfer/receive tasks repeatedly. Both channels also get a status value, to check whether something went wrong during data transfer.

In addition to the existing driver functions, send and receive functions were made. The send packets function creates a number of packets and determines the length of each packet. The length for each packet is the same, 200 bytes. 50×2 bytes for data of the uncorrected frame and 50×2 bytes for the correction values. Since the packets all are of the same size, one block descriptor is enough. As such the block descriptor is stored as a constant, and called upon whenever needed. The packets are sent 32 bits at a time, as this is the width as the AXI DMA.

Before packets are sent, all packets need to have data allocated to them. This is done by reading the correction matrix, all correction values for one corrected sample are taken, so are the uncorrected frame samples that are mapped to the values. This process is repeated 50 times in a for loop. Once the packing is done, the data cache is flushed and the sending process is started.

The receive packets function works similar to the send packets function. Except for the fact that only one packet at a time will be received. Each packet contains a corrected sample, with a size of 16 bits.

The main function calls the setup functions, the system file that contains the cube dimensions

and the correction matrix. The correction matrix is read into three 1-D arrays. Thereafter, a for-loop is initiated, running until all frames are corrected. Inside the for-loop the current uncorrected frame needed for correction is read from the SD-card and put into a 1-D array. An index value indicating which sample is being corrected is updated whenever needed, and then the values for the first correction are sent to the send packets function. Once sending is complete, the receive packets function is called, the corrected sample received is then stored on the SD-card. The main function also keeps track of which corrected values are received. Since not all samples in the corrected frame are being corrected, the main function sets the values not being corrected to zero. This is done by checking the difference between the current and next index of the samples for the corrected frame.

3.3 Verification

Verification was done for the driver, the correction modules and the complete system. The software improvements made to implementation of the feasibility project are tested the same way as in the feasibility project [10]. In short, the obtained results are compared with the results obtained from the Python implementation created by an earlier master student [6].

To test the driver, it was first necessary to create hardware to test it on. The hardware that was used for testing is the same as in the final setup, with the exception for the correction module, which was replaced by a AXI4 data FIFO, see Figure 3.4.

Figure 3.4: Block diagram of test setup

The hardware test setup was first confirmed working by using the AXI DMA test examples, which are provided together with Vivado. Once the functionality of the hardware was tested the driver was tested. This was done by loading the files as normal, and sending the packets as if they would be sent to the corrections core. The difference being that instead of receiving corrected data, the FIFO would simply send the received data back, creating a loop for the transferred data. The CPU then checks the received values against the sent out values, if they match the test succeeds, if they don't match it fails. This way both the software and the hardware that are necessary to drive the correction core is tested. Meaning that if the test succeeds, any potential problems are in the hardware created for the correction core.

The correction modules were tested by sending in known data through a testbench, see Appendix A. The buffer module is tested by imitating the AXI handshake and sending in known data, and checking if the received data is ordered correctly and sent to the multiplication modules. As well as checking if the correct enable signals are sent to the multiplication modules.

The multiplication module is tested by imitating the control signals that enable the multiplication, as well as putting known data on the two input vectors. The output vectors

are checked against expected values that have been computed from the values sent to the input vectors. If the values match, the module works, if they don't match, it does not work.

Finally, the gathering module is tested by imitating the input vectors sent from the multiplication core. Then the output signals are checked, to see if the data sent out matches the data that was put in. In addition to checking the data, the control signals ensuring that sending is done correctly, are checked to see if their status is correct.

For the complete system the test is done in the same way as for the software module. A test file is sent into the system, which then does the corrections and the resulting file is then compared to the file that the Python and C solution produced. If they match, everything works, if not and if the preceding tests passed a problem in the interconnects exists.

3.4 Alternative Attempted Solutions

Other than the final solution, two more other variants were implemented. Both used the correction matrix in the original order in CISR format. The actual hardware was mostly the same, with the sizes of the input and output vectors being different, and the actual multiplication module being different as well.

The first solution attempted to use the CubeDMA to send data to the correction core. This solution would do four corrections at a time, completing the corrections for one uncorrected sample at a time. The multiplication module had four threads that would add together values that were mapped to the same corrected sample. The corrected samples were then sent back. Since samples of the corrected frame are corrected multiple times, after each correction new instances of the corrected frames were saved in memory. Once all a full frame was corrected all corrected samples that were mapped to the same index were added together, completing the correction.

The problem with this solution was that saving new instances for corrected samples would either results in a lack of space in RAM, or the need to read and write to a file. A solution that requires more RAM than available will obviously not work, and by added the amount of reads and writes necessary, this solution would become slower than the C implementation, making it pointless.

The second solution used the AXI DMA with scatter gather, as in the final solution. This time the software driver checked which corrected samples were going to be corrected and added them to the data sent to the correction core. This allowed for only one instance per corrected samples to be kept in memory.

The problem that occurred by doing it this way is that doing the corrections for only one uncorrected sample at a time resulted in being significantly slower than the software implementation. This meant that more corrections had to be done in parallel, which is the point of a programmable logic implementation in the first place. However, the problem with correcting several uncorrected samples at a time is that they might correct the same corrected sample. Meaning that synchronisation and control logic has to be implemented between threads. This was attempted, but was deemed to be too complicated to be finished in the time allowed for this master thesis. As such the final implementation was chosen to be the most viable way of proceeding.

4. Results

This chapter will present the results for the software and FPGA solutions. Time for completion, memory used, how adaptable the solutions are and whether the solutions work at all will be presented here. Thoughts and discussion of the results will be given in chapter 5. All results presented here are based on a hyperspectral cube with the dimensions (row, band, column) of $1088 \times 2048 \times 235$, with a hyperspectral frame of the complete cube being 2048×1088.

4.1 Performance and Resource Requirements of Software Implementation

The software implementation had similar performance to what was estimated in the feasibility study [10]. It now produces correct results, as can be seen from Figure 4.2 and Figure 4.1.

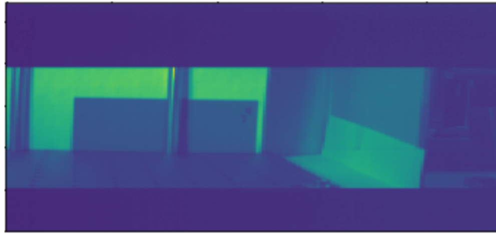


Figure 4.1: Uncorrected hyperspectral cube

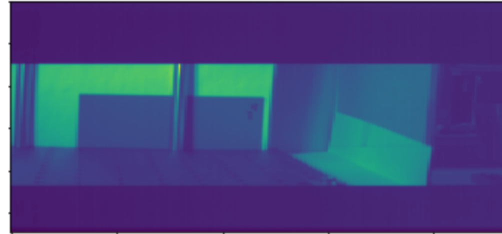


Figure 4.2: Corrected hyperspectral cube

The software solution was tested both on a laptop with an Intel Core i7-8550U Processor, with clock speeds of 1.8 - 4.0 GHz, as well as 16 GB of DDR4 RAM, and the ZedBoard. The tests on the laptop were performed both on Windows 10 and Ubuntu 18.04.3 LTS. With both operating systems yielding the same results. The tests on ZedBoard were done as a stand-alone application, meaning no operating system was used. This was done to limit error sources, with testing on Linux being left for the end if time allowed for it, which was not the case.

Table 4.1 shows the time needed to do the corrections. While Table 4.2 shows how much RAM is being used.

Table 4.1: Performance of software solution

Platform	Hyperspectral cube	Hyperspectral frame
Laptop	$\sim$ 60 s	$\sim$ 255 ms
ZedBoard	$\sim$ 173 s	$\sim$ 736 ms

Table 4.2: Memory usage of software solution

Used for	Uncorr frame	Corr frame	Corr matrix	Other	Total
Memory used	4.46 MB	4.46 MB	53.48	$\sim$ 2.00 MB	$\sim$ 64.40 MB

Since the dimensions of the hyperspectral cube are defined by a header file, the software solution is able to correct cube of various dimensions without the need to change any of the code, only the header file needs to be updated. There are however still some restrictions, the size of the frames to be corrected needs to fit into memory. Considering that the test frame contains more than two million elements and requires less than 100 MB RAM, this is unlikely to become a problem. The other restriction is that the correction matrix needs a frame with dimensions of at least 100×100, frames smaller than that will not be get proper results, again this is a problem that is unlikely to occur.

The generation of the correction matrix for the software corrections has not received any changes, and performs as described in [10]. The performance for the new functions added, namely changing format to CISR and reordering the correction matrix, so that correction is done one corrected sample at a time is shown in Table 4.3.

Table 4.3: Performance of added software functions

	CISR format	Reordering
Time used	$\sim$ 500 ms	$\sim$ 15 minutes
Memory used	$\sim$ 100 MB	$\sim$ 100 MB

4.2 Performance and Resource Requirements of Hardware Implementation

The hardware solution was tested as a stand-alone application running on the ZedBoard. Vivado SDK was used to program the ZedBoard and to upload data to the RAM. Since a stand-alone application lacks a file system, the data needed for correction and the resulting data had to be written to RAM via UART. Since the DDR RAM on the ZedBoard is limited to 512 MB it was not possible to correct a complete cube, rather a single frame was used for testing.

Since the solution on programmable logic completes one corrected sample at a time it makes no difference how much correction is done, as the amount of time spent on subsequent frames will be the same as for the first one. This does however mean that the time spent reading and writing to a file is not included in the test results. The reason a stand-alone application was chosen for testing, is the same as for the software application, to limit sources of errors. Testing on Linux was to be done if time was available at the end, which was not the case. Figure 4.3 and Figure 4.4 respectively show the corrected frame from the proven working software solution and the corrected frame from the hardware solution. As can be seen, the correction module returns a completely black frame, meaning all values are zero, or at least close to. This stems from an error in the outgoing handshake of the correction core, valid data is asserted a clock cycle too early.

Figure 4.3: Corrected hyperspectral frame, corrected in software

Figure 4.4: Corrected hyperspectral frame, corrected on programmable logic

Performance wise Table 4.4 shows the time spent computing and Table 4.5 shows the memory usage of the hardware solution. The hyperspectral cube value is calculated by multiplying the correction time for a frame by 235. As a reminder, these times do not include the time it would take the software driver to read and write from a file.

Table 4.4: Performance of hardware solution

Platform	Hyperspectral cube	Hyperspectral frame
ZedBoard	$\sim$ 276 s	$\sim$ 1.18 s

Table 4.5: Memory usage of hardware solution

Used for	Uncorr frame	Corr frame	Corr matrix	Other	Total
Memory used	4.46 MB	2 B	53.48	∼ 10.00 MB	∼ 68.00 MB

Most of the time spent is used for data transfers, the correction process itself takes only 30 ns, as can be seen from Figure 4.5. The shown simulation only multiplies the values for four corrections, but the amount of additions done made no difference in the time spent calculating.

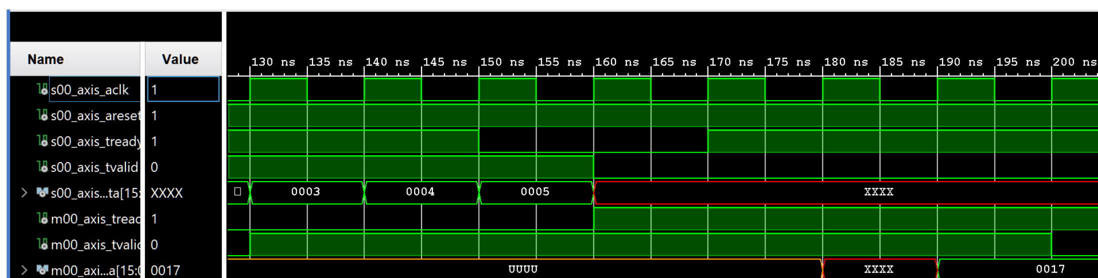


Figure 4.5: Simulated results correction core

Since the software driver determines which corrections are to be done, while the hardware does the corrections, the same header file used in the software solution can be used for this solution. This means that the hardware solution, just as the software solution is able to correct cubes and frames of varying sized without the need to change anything but the header file.

Another benefit of letting the software making the decisions on what data is needed for corrections is that the hardware used for the corrections is actually fairly simple. A FIFO that can easily be resized based on ones needs, a multiplication module that can be extended or shrunk easily and another FIFO that gathers data and sends it back to memory.

Table 4.6 shows how much of the available logic of the ZedBoard is used. The power used by the implementation is 1.75 W, which results in an expected operating temperature of 45°C. Leaving a thermal margin of 3.3 W.

Table 4.6: Area used by hardware solution

	Slice LUTs	Slice Registers	Slice	LUT as logic	LUT as memory
Available	53200	106400	13300	53200	17400
AXI DMA	1833	3403	1059	1726	107
AXI Stream	3349	4193	1273	2641	708
Other	384	505	197	357	59
Correction core	3435	203	1016	883	2552
Total in %	16.9	7.8	26.6	7.6	19.6

5. Discussion

The chapter will discuss the results, assess if the solution works well enough to replace the software solution, consider if things could have been differently and look at future improvements.

5.1 Performance of Hardware

Due to time constraints it was not possible to fully test the hardware implementation. This means that the time required to read and write to a file are not included in the obtained results. A good idea can still be formed based on the results though. The hardware, with only one thread implemented, was outperformed by the software. Even without considering the added time spent for reading and writing. One note that has to be made though is that most of the time was invested in transferring data, the correction itself took no time at all. By reducing the amount of data that needs to be sent, the hardware solution could be made significantly faster. While the current implementation runs on only one thread, adding several threads would make no difference, as the bottleneck is the data transfer.

Since this implementation makes use of the AXI DMA, while the rest of the system uses the CubeDMA, a new bit-stream has to be loaded onto the FPGA before running the smile and keystone corrections. The positive side of this is that all of the programmable logic on the FPGA can be used for this one application.

With decrease in performance and the need to switch bit-streams when wanting to run corrections on hardware, it is hardly worth using over the software implementation. Since there is no point in adding extra cores the only solution is to reduce the amount of data that needs to be sent. This can be done by sending only the needed data for the corrections, instead of the 50 values that are currently sent. However, this would require more work from the software, to determine how many corrections that particular sample needs. The hardware would also need to be more complex, or do multiplication and addition in several rounds. All in all, the result would still likely not be worth using over the software implementation.

Memory usage is similar to the software implementation, to the point that it makes no real difference which solution is used, as far as memory is concerned. The same goes for power usage, Vivado estimates the power usage of the software solution to be 1.70 W, which is slightly less than the hardware solution, but the difference is small enough to not cause problems for the system either way. Adding threads would increase the power consumption of the hardware, but the multiplication module is small enough that even with several threads, the added power consumption is unlikely to cause thermal problems.

As far as adaptability is concerned the hardware solution is able to handle varying frame sized. This is only limited by the fact that the generation of the correction matrix cannot handle frame sizes less than 100×100 and that the DDR3 RAM on the ZedBoard is 512 MB. However, both of these limitations are unlikely to occur. This is because the pictures taken by the HSI are meant to span large areas, and pictures of large areas with few pixels means inaccurate data. As for the RAM limitation, the hyperspectral cube that is used for testing has frame dimensions of 1088×2048 and only takes up roughly 70 MB of space. With the on-board HSI, taking pictures large enough to cause problems with RAM is unlikely, as that would result in a cube with very few frames, meaning only a very small area would be covered.

Another problem the hardware solution incurs is that the generation of the correction matrix with the required reordering takes 15 minutes. While the correction matrix only has to be generated now and then, 15 minutes is still a long time. Also, since the option to capture hyperspectral cubes of varying sizes is wanted, this would mean that 15 minutes has to be spent for every desired dimension. This can quickly turn into a problem, since the SD-card has limited

space, meaning that saving a lot of correction files for different dimension is not really viable.

5.2 Future Work

To make the hardware solution work on the target system, one main task needs to be completed, and that is the Linux driver for the AXI DMA. This was not achieved during this thesis. The stand-alone driver is likely very similar to the eventual Linux driver. The main task would be to update the libraries used by Vivado SDK to Linux compatible libraries. It is also possible that the driver could work on Linux as is, however this has not been tested.

In addition to updating the driver to work on Linux, read and write functions for the file system on the SD-card also have to be added. This part should be rather straight forward and similar to how reading is done in the software solution. The file path just has to be updated to match the locations of the required files on the target system.

Further, the outgoing handshake needs to be corrected, currently the outgoing valid signal goes high to early, resulting in the reading of incorrect or non-existing data.

Once these three tasks have been completed, the hardware implementation should work on the target system.

5.3 Future Improvements

There are several improvements that can be made to the hardware. The first improvement has already been discussed, and that is to reduce the amount of data that has to be sent. This would require a large amount of work, and likely not result in worthwhile results.

Another improvement, that would both simplify the design and remove the problem of the correction matrix generation taking too long, is to map the correction from the corrected frame to the uncorrected frame. In the current implementation the mapping is done from the uncorrected frame to the corrected frame.

By switching the mapping, the attribute of having each uncorrected sample mapped to four corrected samples would be retained, but switched around. Meaning that now, each corrected sample is mapped to four uncorrected samples. Having this attribute makes the correction very predictable and allows for CISR format to be used easily. This way of mapping was tested and proven working with a Python implementation, created by other students and postdoctoral researchers on the HYPSO-team. However, the existence of this method was found and considered too late in the process to make a realistic attempt at implementing it.

By using CISR format, the correction matrix would require less space in RAM, and the multiplication module of the correction core could always do four parallel multiplications for each thread. This means that unlike the current application, were some of the parallel multiplication done might be unnecessary, all multiplication done would be useful. Additionally, since only four, instead of 50, multiplication are done now, a significant amount of area and power can be saved. This saved area could then be used to either add more threads or run other applications on the side. Finally, since only four data sets have to be transferred, the problem with the memory transfer bottleneck would disappear. There would still not be any point in adding additional threads, as even with this little data, the correction is done before new data is ready.

A quick implementation of such a correction module was done and tested, and gave results that were significantly better than the current module. Based on the simulated results, a frame would take roughly 250 ms to correct, with the whole cube taking roughly 60 s. This would end up being on par with the laptop performance of the software solution. This is not as fast the desired 65 ms per frame, but while using the AXI DMA that is likely the fastest implementation that is achievable using the methods explored during this thesis.

Finally, there is one improvement that could be beneficial to all solutions, and that is to increase the data width of the AXI DMA. Currently, it is at 32 bits, the DMA however, supports up to 1024 bit width. During this thesis, widths of more than 32 bits ended up not working correctly. If this problem could be solved the data transfer bottleneck would stop being a problem, and several correction threads could be spawned, thus achieving the desired 65 ms per frame, or better.

5.4 Alternative Implementations

If it should prove absolutely necessary to use the CubeDMA instead of the AXI DMA, there are two possible options, at least based on the work done in this thesis.

The first one would be to use the mapping mentioned in section 5.3 together with the CubeDMA. The CubeDMA does not have scatter gather, which means that the data needed for the corrections has to be either sent over several transactions, or the complete frame has to be streamed each time, with logic being created to sort out the data needed for the current correction. Both options would result in significantly slower corrections, and the latter option would also require additional logic.

The second solution would be to not use a DMA at all, and instead use BRAM. BRAM is a memory resource on the FPGA that can be accessed by both the CPU and the programmable logic. The drawback of this solution is that writing and reading from BRAM is less secure than transfers over a DMA. Data coherency also becomes a problem to consider. Also, since the BRAM of the ZedBoard is only 4.92 Mb, the amount of data that can be stored at a time is limited, which limits the amount of threads that can be driven.

The software driver for the BRAM would be simpler than the driver for the AXI DMA, but the logic required to do access the BRAM would be more complicated. Also, because of the increased chance of errors occurring during writes and some kind of error checking would have to be implemented.

Both solutions would likely be a downgrade in performance over an implementation using the aforementioned mapping with the scatter-gather mode of the AXI DMA.

6. Conclusion

The FPGA solution obtained during this project is able to run on the ZedBoard, which is similar to the on-board hardware of the CubeSat. Since the CPU and RAM on the on-board hardware are the same as on the ZedBoard, there is no reason to believe that the presented solution would be unable to run on the CubeSat hardware.

The performance of the hardware solution, as is, is worse than the performance of the software solution. However, a few simple changes and improvements can be made to create an implementation that should significantly outperform the software solution. This can be achieved by switching how the mapping is done for the correction matrix, so that it now maps from the corrected frame to the uncorrected frame and ordering the correction matrix in CISR format. In addition to this adding more threads and increasing the AXI DMA data width, allowing for several samples to be corrected at the same time, should result in an implementation that achieves the desired performance of at least 15 FPS. As far as discovered during this thesis, limitation on data transfers is a big problem, while area is not a problem at all.

The scatter-gather mode of the AXI DMA turned out to be essential in creating an effective solution. While the CubeDMA could have been used, the resulting performance would have been worse than the performance of the current solution and the software solution. As such, even though the rest of the system uses the CubeDMA, the AXI DMA was used for the implementation created for this thesis.

For both the hardware and software solution testing on Linux remains. The hardware solution also requires a fix for the outgoing handshake. Testing on the actual hardware of the CubeSat also has to be done, as testing during this thesis was done only on the ZedBoard, which has similar specifications.

All in all, while the final solution did not achieve the desired performance, a way to improve and achieve acceptable results was found. While the mentioned improvements and changes have not been tested, the changes that need to be made, are small enough that there is no reason to believe that it won't work.

References

[1] A. Varntresk. Assembly and testing of baseline processing chain, 2019.

[2] NanoAvionics. 6U nanosatellite bus M6P. https://nanoavionics.com/nanosatellite-buses/6u-nanosatellite-bus-m6p/ [26.06.20], 2019.

[3] A. V. Nytrø. Mct depot: An open mct telemetry data visualisation system for ntnu hypso smallsat mission and operations, 2020.

[4] N. Yokoya, N. Miyamurab and A. Iwasaki. Preprocessing of hyperspectral imagery with consideration of smile and keystone properties, 2010.

[5] M. T. Eismann. Hyperspectral Remote Sensing, 2012.

[6] M. B. Henriksen, J. L. Garret, E. F. Prentice, A. Stahl and T. A. Johansen. Real-time corrections for a low-cost hyperspectral instrument, 2019.

[7] Johan Fjeldtvedt. Testing of Communication between Various Peripherals on ZED-Board, 1991.

[8] XILINX. AXI DMA v7.1 LogiCORE IP Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf [26.06.20], 2019.

[9] Nasa. CubeSat 101 Basic Concepts and Processes for First-Time CubeSat Developers . https://drive.google.com/file/d/1hy164MsNWN1gP_HLoxX7XrcSYjWO0q-B/view [26.06.20], 2017.

[10] M. Montzka. Fast Spectrograph Corrections, 2019.

[11] R. A. Schowengerdt. Chapter 9: Thematic Classification, in Remote Sensing - Models and Methods for Image Processing, 2007.

[12] F. Sigernes. Basic Hyper Spectral Imaging, Lecture notes NTNU. http://kho.unis.no/doc/TTK20/BASIC_HYPER_SPECTRAL_IMAGING.pdf [30.10.19], 2018.

[13] S. Nicolas T. Opsahl T. Haavardsholm I. Kåsen T. Skauli, H. E. Torkildsen and A. Rognmo. *Compact Camera for Multispectral and Conventional Imaging Based on Patterned Filters*. Applied Optics, vol. 53, no. 13, pp. C64–C71, 2014.

[14] P. Mouroulis and M. M. McKerns. *Pushbroom imaging spectrometer with high spectroscopic data fidelity: experimental demonstration*. Optical Engineering, vol. 39, no. 3, pp. 808–816, 2000.

[15] Y. Wang H. Yuan H. Feng B. Xu G. Yang, C. Li and X. Yang. *The DOM Generation and Precise Radiometric Calibration of a UAV-Mounted Miniature Snapshot Hyperspectral Imager*. Remote Sensing, vol. 9, no. 642, pp. 1–21, 2017.

[16] T. G. Chrien V. Duval R.O. Green J. J. Simmonds P. Mouroulis, D. A. Thomas and A. H. Vaughan. *Trade Studies in Multi/hyperspectral Imaging Systems Final Report*. tech. rep., Jet Propulsion Laboratory, 1998.

[17] M. B. Henriksen. Hyperspectral Imager Calibration and Image Correction, 2019.

[18] T. A. Davis. *Direct Methods for Sparse Linear Systems*. siam, 2006.

[19] A. Jain N. Goharian and Q. Sun. *Comparative analysis of sparse matrix algorithms for information retrieval.* Computer, vol. 2, pp. 0–4, 2003.

[20] J. Fowers, K. Ovtcharov, K Strauss, E. S. Chung, and G. Stitt. *A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication.* IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, vol. 22, pp. 36–43, 2014.

[21] M. Danielsen. System integration and testing of on-board processing system for a hyperspectral imaging payload in a cubesat, 2020.

[22] ARM. AMBA 4 AXI4-Stream Protocol. https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf [26.06.20], 2010.

[23] J. A. Gjersund. A reconfigurable fault-tolerant on-board processing system for the hypso cubesat, 2020.

[24] AVNET. ZedBoard (Zynq™Evaluation and Development) Hardware User's Guide. http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf [26.06.20], 2014.

[25] XILINX. Zynq-7000 SoC Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf [26.06.20], 2018.

Appendix

A. Testbench Correction Modules

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 use std.env.finish;
6
7 entity axi_fifo_tb is
8 end axi_fifo_tb;
9
10 architecture sim of axi_fifo_tb is
11
12   -- Testbench constants
13   constant clock_period : time := 10 ns;
14   constant C_AXIS_TDATA_WIDTH : natural := 32;
15   constant ADDR_WIDTH : natural := 100;
16
17 -- AXI slave interface
18     signal s00_axis_aclk : std_logic :='1';
19     signal s00_axis_aresetn :  std_logic :='0';
20     signal s00_axis_tready :  std_logic;
21     signal s00_axis_tvalid :  std_logic :='0';
22     signal s00_axis_tdata :  std_logic_vector(C_AXIS_TDATA_WIDTH - 1 downto 0) :=
       (others => '0');
23     signal s00_axis_tstrb :  std_logic_vector((C_AXIS_TDATA_WIDTH/8) - 1 downto 0)
       ;
24     signal s00_axis_tlast :  std_logic;
25
26     -- AXI master interface
27     signal m00_axis_aclk :  std_logic;
28     signal m00_axis_aresetn :  std_logic;
29     signal m00_axis_tready :  std_logic:='0';
30     signal m00_axis_tvalid :  std_logic;
31     signal m00_axis_tdata :  std_logic_vector(C_AXIS_TDATA_WIDTH - 1 downto 0);
32     signal m00_axis_tstrb :  std_logic_vector((C_AXIS_TDATA_WIDTH/8) - 1 downto 0)
       ;
33     signal m00_axis_tlast :  std_logic;
34
35
36 begin
37
38   s00_axis_aclk <= not s00_axis_aclk after clock_period / 2;
39
40   DUT : entity work.axi_fifo(rtl)
41   generic map (
42     C_AXIS_TDATA_WIDTH => C_AXIS_TDATA_WIDTH,
43     ADDR_WIDTH => ADDR_WIDTH
44   )
45   port map (
46     s00_axis_aclk => s00_axis_aclk,
47     s00_axis_aresetn => s00_axis_aresetn,
48     s00_axis_tready => s00_axis_tready,
49     s00_axis_tvalid => s00_axis_tvalid,
50     s00_axis_tdata => s00_axis_tdata,
51     s00_axis_tstrb => s00_axis_tstrb,
52     s00_axis_tlast => s00_axis_tlast,
53     m00_axis_aclk => m00_axis_aclk,
54     m00_axis_aresetn => m00_axis_aresetn,
55     m00_axis_tready => m00_axis_tready,
56     m00_axis_tvalid => m00_axis_tvalid,
57     m00_axis_tdata => m00_axis_tdata,
58     m00_axis_tstrb => m00_axis_tstrb,
59     m00_axis_tlast => m00_axis_tlast
60   );
61
62
63   PROC_SEQUENCER : process is
64   begin
```

```
65
66      wait for 10 * clock_period;
67      s00_axis_aresetn <= '1';
68      wait until rising_edge(s00_axis_aclk);
69
70      report "Writing";
71
72      -- Write until full
73      s00_axis_tvalid <= '1';
74      while s00_axis_tready = '1' loop
75        s00_axis_tdata <= std_logic_vector(unsigned(s00_axis_tdata) + 1);
76        wait until rising_edge(s00_axis_aclk);
77      end loop;
78      s00_axis_tvalid <= '0';
79
80      s00_axis_tdata <= (others => 'X');
81
82      report "Reading";
83
84      -- Read until empty
85      m00_axis_tready <= '1';
86      while m00_axis_tvalid = '1' loop
87        wait until rising_edge(s00_axis_aclk);
88      end loop;
89      m00_axis_tready <= '0';
90
91      report "Test completed.";
92      finish;
93    end process;
94
95  end architecture;
```

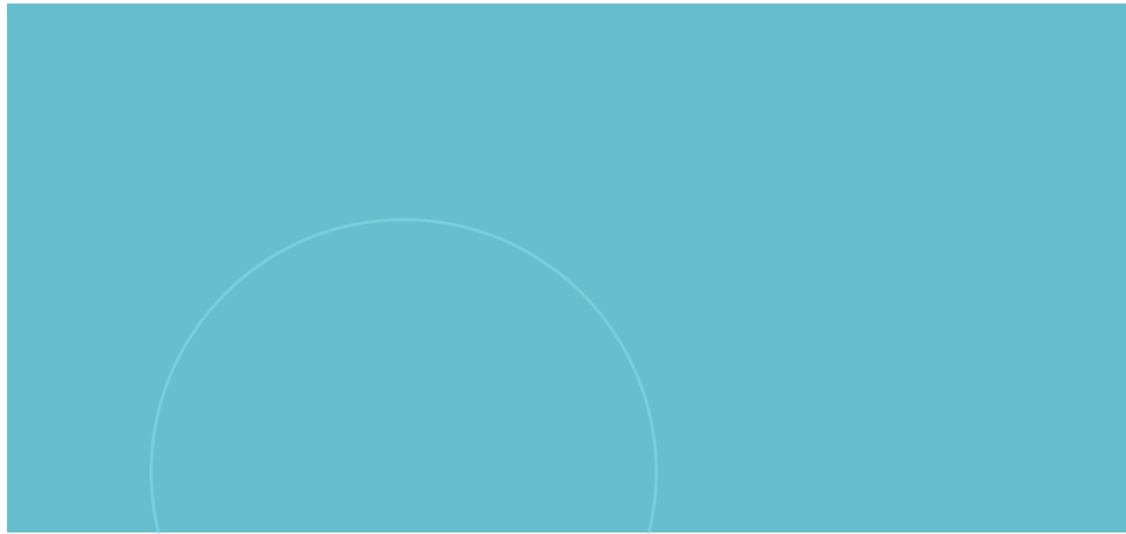
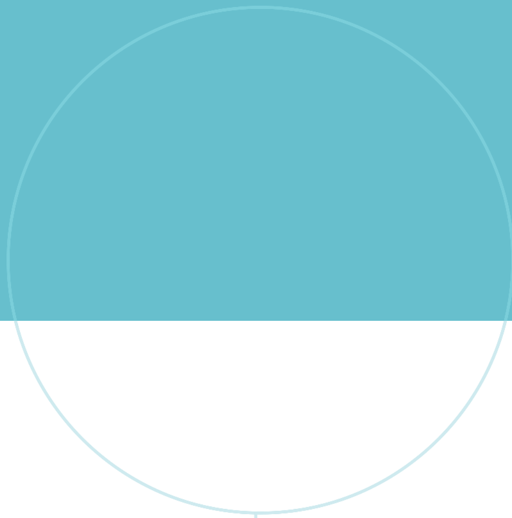
B. Register overview CubeDMA [1]

Field	Description	Bits
Register 0x00		
Start	Starts when this bit transitions from 0 to 1	0
Blockwise mode	Cube is read in blocks	2
Planewise mode	Cube is read in plains	3
Error IRQ enable	IRQ when error occurs	4
Completion IRQ enable	IRQ when transfer completed	5
Number of plane transfers	ceil(depth/C_MM2S_NUM_COMP)	8-15
Start offset	Number of components start offset	16-23
Register 0x04		
Transfer done	Indicates that transfer has completed	0
Error code	Indicates error condition(s) that occured during transfer	1-3
Error IRQ flag	On Read: 1 when IRQ triggered due to error On Write: Writing 1 clears the IRQ flag	4
Completion IRQ flag	On Read: 1 when IRQ triggered due to completion On Write: Writing 1 clears the IRQ flag	5
Register 0x08		
Base address	Address of the first component in the HSI cube	0-31
Register 0x0C		
Width	The width of the HSI cube	0-11
Height	The height of the HSI cube	12-23
Depth (low)	Lower 8 bits of the depth / number of planes of the HSI cube	24-31
Register 0x10		
Block width	log2 of block width in pixels	0-3
Block height	log2 of block height in pixels	4-7
Depth (high)	High 4 bits of the depth / number of planes of the HSI cube	8-11
Last block row size	Size of one row in the last block in each row of blocks	12-31
Register 0x14		
Row size	Number of components in one row of the cube	0-19

Field	Description	Bits
Register 0x20		
Start	Core starts transfer when this bit transitions from 0 to 1	0
Error IRQ enable	Trigger IRQ when error condition arises	4
Completion IRQ enable	Trigger IRQ when transfer is complete	5
Register 0x24		
Transfer done	Indicates that transfer has completed	0
Error code	Indicates error condition that occured during transfer	1-3
Error IRQ flag	On Read: 1 when IRQ triggered due to error On Write: Writing 1 clears the IRQ flag	4
Completion IRQ flag	On Read: 1 when IRQ triggered due to completion On Write: Writing 1 clears the IRQ flag	5
Register 0x28		
Base address	Address of where to store data	0-31
Register 0x2C		
Received length	Number of bytes received from start of transfer until TLAST was asserted	

-

Name	Description
C_MM2S_AXIS_WIDTH	Bit width axi stream from memory
C_MM2S_COMP_WIDTH	Bit width for one sample from memory
C_MM2S_NUM_COMP	Number of components in one word from memory
C_S2MM_AXIS_WIDTH	Bit width axi stream to memory
C_S2MM_COMP_WIDTH	Bit width for one sample to memory
C_S2MM_NUM_COMP	Number of components in one word to memory
C_TINYMOVER	Use of tinymover or datamover

NTNU
Norwegian University of
Science and Technology