Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Karine Avagian

An FPGA-oriented Hardware Implementation of Richardson-Lucy Deconvolution Algorithm for Hyperspectral Images

Masteroppgave i Elektronisk Systemdesign og Innovasjon
Veileder: Kjetil Svarstad, Milica Orlandic
Juni 2019

NTNU
Kunnskap for en bedre verden

Karine Avagian

An FPGA-oriented Hardware Implementation of Richardson-Lucy Deconvolution Algorithm for Hyperspectral Images

Masteroppgave i Elektronisk Systemdesign og Innovasjon
Veileder: Kjetil Svarstad, Milica Orlandic
Juni 2019

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

NTNU
Kunnskap for en bedre verden

(This page is intentionally left blank)

Abstract

This work presents a method to reduce the spatial degradation in hyperspectral images caused during the image acquisition process. The degradation is modeled by a convolution with a *Point Spread Function* (PSF), which in this work, is assumed to be known. The three-dimensional hyperspectral images are modeled as a composition of two-dimensional independent images. Degradation is reduced by applying an accelerated Richardson-Lucy (RL) deconvolution algorithm on each individual image. Boundary conditions are introduced in order to keep a constant image size without distorting the estimated image boundaries. An algorithm is implemented in C in both the floating-point and fixed-point representations. The quantization error between the two representations is negligible. The RL-deconvolution algorithm is fully ported on an FPGA-based platform (i.e., Xilinx Zynq-7020) using the hardware description language VHDL. Two architectures are designed and called *Architecture-1* and *Architecture-2*. The former is optimized with respect to the communication time with an external memory and the latter is optimized for limited storage. Both architectures are parameterized with respect to the image size and run-time configurable with respect to the number of iterations. In addition, Architecture-2 is run-time configurable with respect to the kernel size with a maximum kernel size equal to 9×9. The execution time of implemented architectures is compared to a software only implementation of the algorithm running on the same Xilinx Zynq platform. A speed-up by a factor of 31 is achieved for Architecture-1 and a speed-up by a factor of 61 is achieved for Architecture-2. The execution time is also compared to a HW/SW implementation of the RL-deconvolution and a speed-up by a factor of 13 is achieved for Architecture-1 and a speed-up by a factor of 26 is achieved for Architecture-2. Compared to a state-of-the-art solution, a speed-up by a factor of 1.8 is achieved for the Architecture-2 when running a standard RL-deconvolution.

Sammendrag

Denne masteroppgaven presenterer en metode for å forbedre romlig oppløsning i hyperspektrale bilder. Uskarpheter i bilder modelleres ved hjelp av foldning (convolution) med en Point Spread Function (PSF), som antas å være kjent. Et tre-dimensjonalt hyperspektralbilde modelleres som en komposisjon av flere to-dimensjonale bilder. Uklarhet i bildene reduseres ved hjelp av en akselererert Richardon-Lucy (RL) dekonvolusjon. Original bildestørrelse bevares ved hjelp av "boundary conditions". Algoritmen er implementert i C, med både "floating-point" og "fixed-point" representasjoner. Kvantiseringsfeilen mellom representasjonene er neglisjerbar. Algoritmen er derreter beskrevet i VHDL og kjørt på an FPGA-basert plattform (Xilinx Zynq-7020). To arkitekturer har blitt designet, den første, Architecture-1, optimalisert med hensyn til kommunikasjonstid med ekternt minne, og den andre, Architecture-2, optimalisert med hensyn til intern FPGA lagringsplass. Begge arkitekturene er parametrisert med hensyn til bildestørrelse og konfigurerbar i kjøretid med hensyn til antall iterasjoner i algoritmen. I tillegg er Architecture-2 konfigurerbar ved kjøretid med hensyn til kernel størrelse, der maksimal kernel størrelse er 9×9. Kjøretid til Architecture-2 er to ganger raskere en for Architecture-1. Kjøretid for de implementerte arkitekturene er sammenlignet med en software-basert implementasjon som kjøres på samme platform, der akselerasjonen er lik 61 for Architecture-2. Kjøretid er også sammenlignet med en HW/SW implementasjon av RL-dekonvolusjon or en akselerasjon lik 26 er oppnådd for Architecture-2. Sammelignet med en state-of-the-art implementasjon, akselerasjon lik 1.8 for Architecture-2.

Acknowledgments

I would love to thank my co-supervisor Milica Orlandić for her support throughout the entire year and that extra time spent re-reading and editing my first conference paper. This was a small accomplishment I am happy to have. Furthermore, I would like to thank all the guys and girls at the HYPSO project, and a special thanks to Dordje Bošković for sharing the ideas, driving carefully and being as stressed as I am. Lastly, I would like to thank my dearest Steinar, for all his love and care, and my family, for their encouragement and patience.

Now, I can finally pass the torch to my dearest Gabi, enjoy the coming years at NTNU!

Contents

List of Tables

List of Figures

Abbreviations

HIS	=	Hyperspectral imaging system
HSI	=	Hyperspectral image
FOV	=	Field-Of-View
RL	=	Richardson-Lucy
BSNR	=	Blurred Signal-to-Noise Ration
RRE	=	Relative Reconstruction Error
PSNR	=	Peak Signal-to-Noise Ration
MSE	=	Mean Squared Error
SSIM	=	Structural Similarity Index
1-D	=	One Dimensional
2-D	=	Two Dimensional
BC	=	Boundary Condition
Z-BC	=	Zero Boundary Condition
VC-BC	=	Variable Constant Boundary Condition
MP-BC	=	Modified Mirror Boundary Condition
FFT	=	Fast Fourier Transform
FPGA	=	Field-Programmable Gate Array
ASIC	=	Application Specific Integrated Circuit
CPU	=	Central Processing Unit
SoC	=	System On Chip
PS	=	Processing System
PL	=	Programmable Logic
DMA	=	Direct Memory Access
IRQ	=	Interrupt Request
GP	=	General Purpose
HP	=	High Performance
RAM	=	Random Access Memory
BRAM	=	Block Random Access Memory
ROM	=	Read-Only Memory
FIFO	=	First In-First Out
DSP	=	Digital Signal Processing

Chapter 1

Introduction

1.1 Remote Sensing

In 1972, for the first time in history, a general-purpose satellite image data set was captured by the Landsat Multispectral Scanner System (MSS) [6]. The images were composed of only four spectral bands, but more advanced and complex descendants of the MSS followed. Today there exists a wide variety of remote sensing systems for *Earth Observations* (EO) with applications ranging from weather forecasting to military operations. One important application of EO is early detection of natural- or human-induced hazards. An example of the latter could be oil spills, which if not detected early can have catastrophic effects on the environment. Also, natural catastrophes, such as *blooming algae*, which is harmful for both marine life and humans [7], can be detected with the help of remote sensing.

1.1.1 Hyperspectral Imaging

Hyperspectral Imaging System (HIS) is the technology that is used in , e.g., the detection of the toxic algae blooms. HIS combines *spectroscopy*, the study of the interaction between electromagnetic waves and matter, and digital *imaging*, thus providing both spectral and spatial information. An image is formed by a spectrometer, which splits, with a prism or a grating, the detected light, *radiance*, into narrow *spectral bands*, each corresponding to a certain wavelength [8]. Different surfaces interact with the light differently, therefore the amount of the detected radiance differs depending on the surface. In order to characterize a measurement, one can either plot the detected radiance or the measured *reflectance* as a function wavelength. Radiance have units of $[W/m^2/sr/\mu m]$

and it includes illumination, the measurements position and atmospheric effects, while reflectance is the ratio of reflected radiation to incident radiation and describes the intrinsic property of the material [8]. Both terms are often used interchangeably and called *spectral signature* [9].

Figure 1.1 is a composition of three spectral bands taken from the hyperspectral dataset [2], which depicts the coastal line of the part of the New Zealand coast. A light blue area close to the coast depicts the blooming algae. This can also be seen in the radiance plot shown in Figure 1.2, where the red plot is estimated from the area close to the coast, while the blue plot is estimated from the area in the water far from the coast. The spectral classification is done by the comparison to the similar data in [10].

Figure 1.1: A three color-composite of three spectral bands taken from the hyperspectral dataset [2]. Image shows the coast of New Zealand recorded with *Hyperspectral Imager for the Coastal Ocean* (HICO) spectrometer.

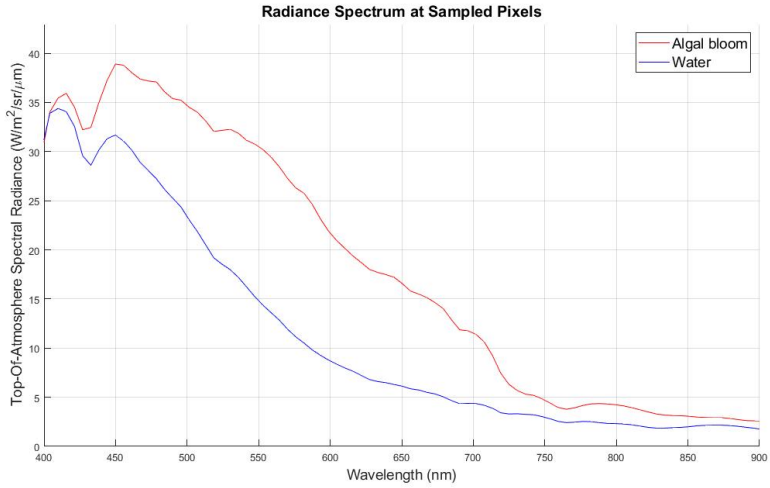


Figure 1.2: Spectral reflectance plot at two samples, one close to the coast and the one farther away from the coast.

1.1.2 Image Acquisition

Hyperspectral images, as the one shown in Figure 1.1, can be acquired by placing the HIS on either an aircraft or a satellite. There are several types of remote sensor systems designed to acquire hyperspectral images. One particular example, is a *pushbroom* scanner, which has a linear array of detector elements, marked gray in the left illustration in Figure 1.3, placed perpendicularly to in-track travel direction of e.g., the satellite. The observed scene is collected slice by slice at each time instance, by moving the sensor or its *Field-Of-View* (FOV) across the scene. A slice, shown in the right illustration in Figure 1.3, refers to *two-dimensional* (2-D) data with width equal to the number of detector elements in the cross-track direction and depth equal to the number of spectral bands.

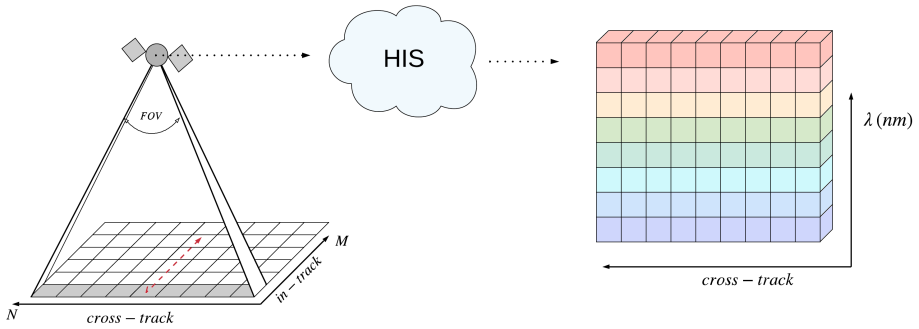


Figure 1.3: An illustration of an image acquisition by a pushbroom scanner. The scanner moves in-track in the direction marked by red arrow, scanning a full width of an image per time instance. The hyperspectral imaging system mounted on the scanner then transforms the collected light into a 2-D matrix with one spatial dimension and one spectral dimension.

The process results is a three dimensional data with *spatial*, *temporal* and *spectral* axis. The spatial and temporal dimensions are measured along the cross- and in-track directions respectively and the spectral dimension is measured along the spectral axis. The recorded *Hyperspectral Image* (HSI) can also be referred to as a *hyperspectral data cube* denoted $\mathbf{Y} \in \mathbb{R}^{N \times M \times P}$, where N and M are the number of spatial and temporal measurements respectively, and P is the number of spectral measurements. The hyperspectral image can be characterized in both the spectral and spatial domain. In the spatial domain, values of samples in one spectral band form a grayscale image with two spatial dimensions, (x, y), where each image depicts the same scene at different, well-defined, wavelength, λ. A whole cube $\mathbf{Y}$ can then be modelled as a stack of two-dimensional spatial images denoted $\mathbf{Y}^p \in \mathbb{R}^{N \times M}$, $p = 1, \ldots, P$. In the spectral domain, a spectral *pixel*, is a vector of size P. Each pixel represents a contiguous spectrum of the reflected radiation for a specific measurement. An illustration of a hyperspectral cube, together a spectral and spatial representation, is shown in Figure 1.4.

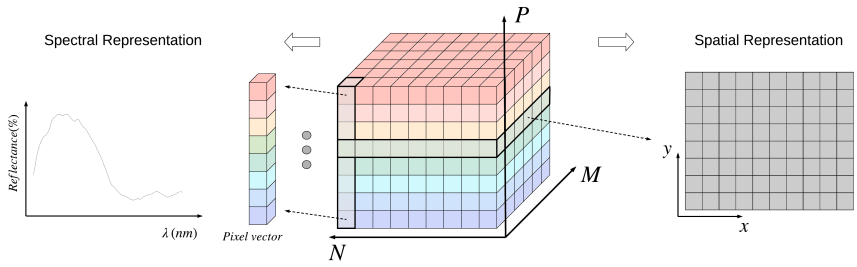


Figure 1.4: An ideal representation of a three-dimensional hyperspectral data cube: (a) a pixel vector and its spectral signature, (b) a hyperspectral data cube and (c) grayscale image for one spectral band.

1.1.3 Radiometric, Spatial and Spectral Resolution

Detector elements collect and store the light reflected by the surface by converting the continuous data stream to quantised and sampled data. Detected by the sensor data is continuous in both with respect to the cross- and in-track directions and in amplitude [11]. When performing sampling, the coordinate values become discrete, and in order to digitalize the amplitude, quantization is performed. Each sample is then defined by a finite number of bits, Q, where Q determines the *radiometric resolution* of the image. The larger the Q, the better is the approximation of the analog signal. In addition to the radiometric resolution, the hyperspectral imaging systems are also classified by their *spectral* and *spatial* resolutions. The spatial resolution refers to the minimum distance at which two different objects are distinguishable. In remote sensing, a common measure of spatial resolution is the *Ground Sampling Distance* (GSD), which refers to the size of the grid elements projected onto the Earth. For example, the hyperspectral imager, HICO, used to acquire the image shown in Figure 1.1 has spatial resolution equal to 90 m, meaning that an object smaller $<$ 90 m will appear in the digital image blended with the surrounding area. The spectral resolution refers to the minimum distance between the recorded bands. For the same hyperspectral image in Figure 1.1, the spectral resolution is 5.7 nm. Some of the HICOs disadvantages are its size and cost. In order to reduce both of them, a trade-off between spectral and spatial resolutions needs to be accounted for. As the spectral information is the main concern when using the hyperspectral imaging, the HIS are often built having a high spectral resolution and a lower spatial resolution. Therefore, post-processing of the data is often inevitable.

1.2 HYPSO mission

This master thesis is written as a part of the HYPSO[1] mission. The main goal of the mission is to observe ocean phenomena, including algae bloom, by the use of an on-board processing chain for HIS. The hyperspectral data is going to be acquired with a hyperspectral imaging system (V6) [12] which has a high spectral resolution equal to 1.67 nm, but low spatial resolution equal to 300 m. A planned data processing pipeline is shown in a block diagram in Figure 1.5. The hyperspectral data cube is acquired and stored in the *Double Data Rate* 3 (DDR3) SDRAM. Then, the hyperspectral data in DDR3 can be streamed efficiently using a special-purpose *Direct Memory Access* (DMA) core, called Cube DMA, developed by J.A.Fjeldtvedt [13]. The data processing starts with the first FPGA accelerator, called *Binning*, which adds two or more spectral bands together. The resulting data is sent to the $Super - Resolution$ block, where the spatial resolution of a whole data cube is enhanced. This step is implemented in this thesis. After Super-Resolution, the hyperspectral data cube is ready to be analyzed with respect to the target detection. Some groundwork regarding the dimensionality reduction for target detection is made by S. Bakken [14] and an efficient HW/SW Implementation of Hyperspectral Target Detection Algorithm is implemented by D. Bošković [15]. Finally, the hyperspectral data cube is compressed in *LosslessCompression* block, before it is transmitted to the ground station. For data compression purposes, the CCSDS-123 Compression Algorithm implemented by J.A.Fjeldtvedt [16] [17] is used.

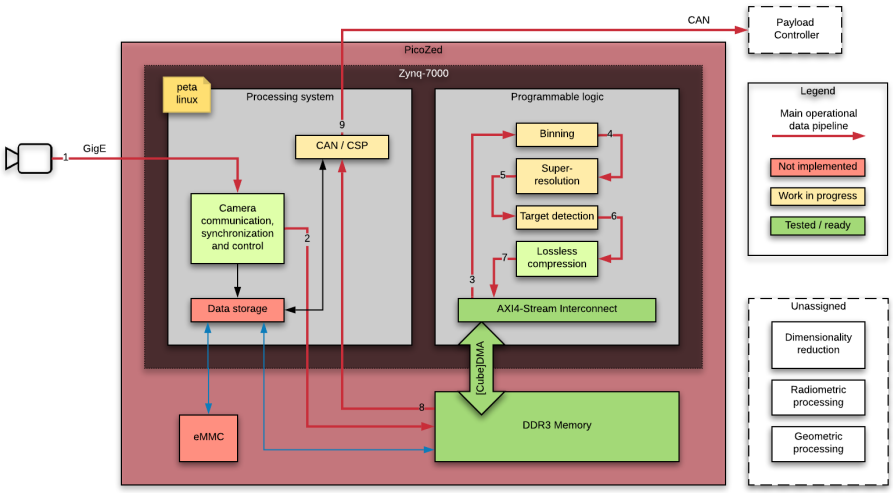


Figure 1.5: On-board data processing pipeline [].

[1] Hyper-spectral SmallSat for ocean Observation

1.2.1 Super-Resolution

The aim of super-resolution is to improve the spatial image resolution by combining information from several low resolution images into one high resolution image [18]. There are two main steps involved in super-resolution: image registration and deblurring. By performing image registration, the pixel information from several low resolution images are placed into the correct positions in a high resolution grid. The deblurring step removes the blur that occurres during the image formation. Deblurring can be done with a technique called deconvolution, which is the main goal of this thesis, and is going to be discussed in more detail in Chapter 2.

In this work, the HYPSO mission requirements [19] for the spatial resolution are used as a motivation. The requirements state that the GSD should be smaller than 100 m is used as a motivation. The image size is equal to $500 \times 1200 \times 100$, with 16 bits data width.

1.2.2 On-Board Computer

The main part of the satellite payloads on-board computer is the PicoZed *System on Module* (SoM) based on the Xilinx Zynq-7000 All Programmable *System on Chip* (SoC). In general, the Zynq architecture comprises a *Processing System* (PS), and *Programmable Logic* (PL). The block diagram for the Zynq SoC is shown in Figure 1.6. The main features of the PS section of the chip in regards to this project, are a dual-core ARM Cortex-A9 processor along with an on-chip memory, external memory interfaces and a set of peripheral connectivity interfaces. The PL section is composed of general purpose FPGA logic fabric, i.e., the *Configurable Logic Blocks* (CLBs) and *Input/Output Blocks* (IOBs). In addition, there are two dedicated and optimized special purpose components, *Block Random Access Memories* (BRAMS) and DSP48E1, designed for storing large amount of data and for high-speed arithmetic, respectively. One BRAM can store up to 36 Kb of data. Both the BRAM and the DSP48E1 can be clocked at the maximum clock frequency of the device.

As shown in Figure 1.6, processing system is connected to the programmable logic using the *Advanced eXtensible Interface* (AXI) ARM AMBA *interconnects* and *interfaces*. The two important AXI interfaces are, a *General Purpose* (GP) AXI, which is a 32-bit data bus for low to medium rate communication, and a *High Performance* (HP) port, which supports a high rate communication. There are three types of AXI4 interfaces, i.e., AXI4-memory-mapped, AXI4-Lite and AXI4-Stream. The AXI4-Lite and AXI4-Stream are discussed in the following section.

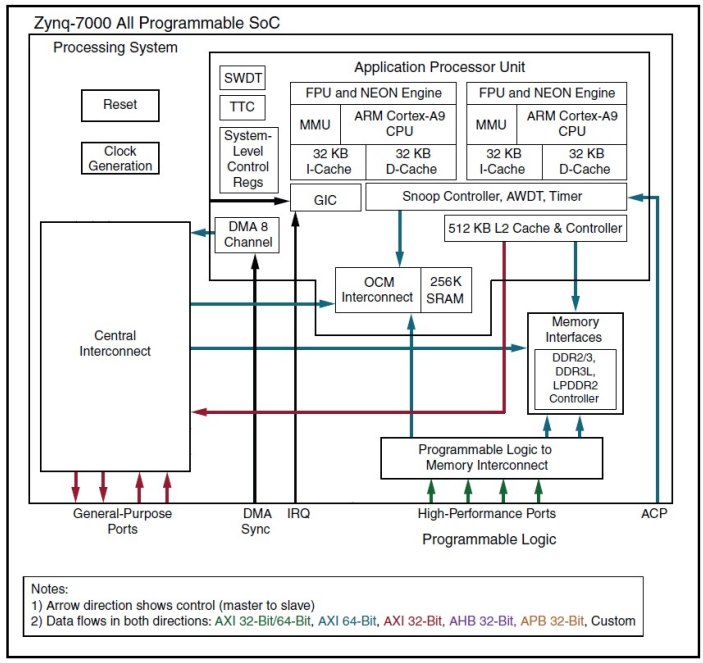


Figure 1.6: Xilinx Zynq-7000 CPU/FPGA SoC block diagram. A modifies image taken from [1].

1.2.3 AXI4 interfaces

AXI4-Lite protocol

AXI4-Lite is used for simple, low-throughput memory-mapped communication [3], between a master port (i.e., the one initiating a transaction) and a slave port (i.e., the one accepting the transaction). AXI4-Lite interface consists of five different channels: Read Address, Write Address, Read Data, Write Data and Write Response. A Read transaction uses the READ address and READ data channels as shown in Figure 1.7 and a Write transaction uses Write address, Write data and Write response channels, shown in Figure 1.8. Write and Read transactions have unique addresses, which allows simultaneous data transfer.

An interaction between a master port and a slave port happens only after a valid/ready *handshake*. Each channel (i.e., address, data and response) has its own handshake. For most of the transactions, except the read transaction, the master port asserts the valid signal and the slave answers with the ready signal. For the read transaction, the slave asserts valid signal, indicating that the validity of the returning data. The AXI4-Lite interface is usually used for control signals

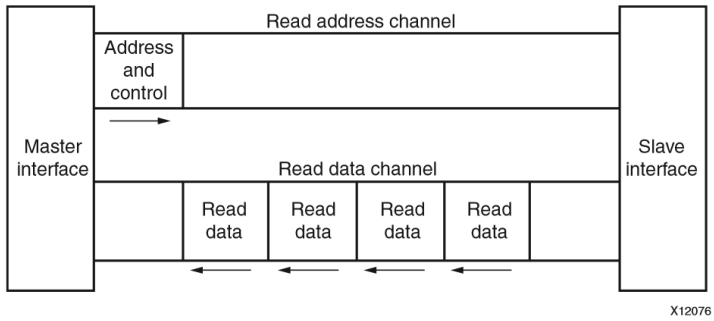


Figure 1.7: Read transaction architecture [3].

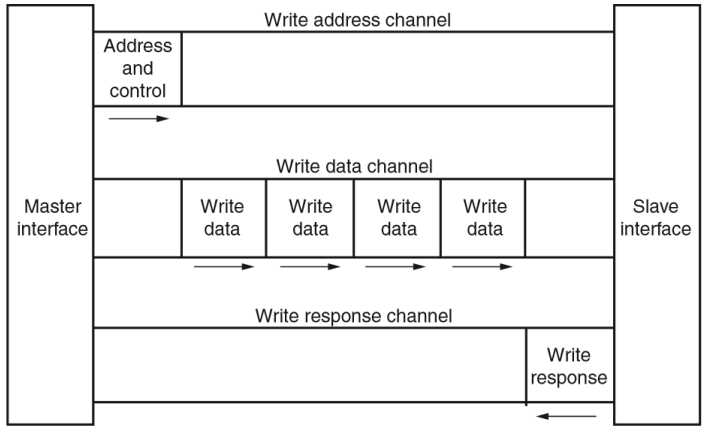


Figure 1.8: Write transaction architecture [3].

and status registers.

AXI4-Stream protocol

AXI4-Stream protocol is used to connect a master module, which generates data, to a slave module, which receives data. Data streaming occurs in a point-to-point fashion, without any addressing, thus the overhead time is reduced. The commonly implemented interface signals are the TVALID, TREADY, TDATA, TLAST, ACLK and ARESETn. Signals TKEEP and TSTRB are auxiliary. The transfer between a master and a slave is controlled by a handshake process. For a transfer to be valid both TVALID and TREADY signals must be asserted to an active HIGH. TVALID is asserted by the master module and TREADY is asserted by the slave module. Either TVALID or TREADY can be asserted first,

or both can be asserted at the same time. It is important that a master module does not wait for the slaves TREADY signal. Also, if TVALID is asserted it should stay asserted until the handshake occurs [3]. All input signals are sampled on the rising edge of ACLK and all output signal changes must occur after the rising edge of ACLK. ARESET is an active-LOW reset signal.

1.3 Main contributions

This thesis presents a fast and reliable deconvolution algorithm implemented on FPGA. Prior to this thesis, during the autumn of 2018, a specialization project was conducted related to the selection of a suitable algorithm. The RL-deconvolution algorithm was chosen due to its robustness against small errors in kernel estimations and overall good visual deconvolution results. A Hardware/Software codesign solution was implemented on a target FPGA. For this project, the whole RL-deconvolution algorithm is ported onto the FPGA. Two architectures are designed, one which after being initializes can operate independently of the PS and the other one which is optimized with respect to the internal storage of FPGA. The implemented architectures are scalable with respect to the image size and run-time configurable with respect to the number of RL-deconvolution iterations. In addition, one of the architectures is designed to handle, at a run-time, kernels of varying size, with a maximum size 9×9.

In addition to the hardware architecture, some additional theoretical analysis of the RL-deconvolution algorithm is done. It is found that the standard RL-deconvolution can be accelerated without a significant increase in resource usage. This is done by adding one additional multiplier. Another aspect of the analysis is the preservation of the image size. RL-deconvolution is based on the convolution operation, where the input image needs to be extended in order to produce an output image of the same size as the input image. The extended image borders are estimated using boundary conditions, which in this project area set to be equal to the first sample value of the input image. The estimated sample changes value for each new RL-deconvolution iteration, thus adapting to the estimated devolved image.

Two papers based on this work have been submitted:

- K. Avagian, M. Orlandić, T. A. Johansen. An FPGA-oriented HW/SW Codesign of Lucy-Richardson Deconvolution Algorithm for Hyperspectral Images. *8th Mediterranean Conference on Embedded Computing – MECO* Montenegro, 2019
- J. L. Garrett, D. Langer, K. Avagian, A. Stahl. Accuracy of super-resolution for hyperspectral ocean observations. 2019

1.4 Thesis structure

The thesis is divided into following chapters:

- Chapter 2 gives a background information regarding image processing, including image formation process and image restoration. This chapter also presets the state-of-the-art implementations of the RL-deconvolution algorithm.
- Chapter 3 presents a theoretical analysis of RL-deconvolution algorithm for hyperspectral images.
- Chapter 4 describes two RL-deconvolution hardware architectures.
- Chapter 5 presents a verification method used to test the implemented designs.
- Chapter 6 presents the synthesis results, including resource utilization, power estimation and execution time estimations for two architectures of RL-deconvolution algorithm. The results are also discussed.
- Chapter 7 draws the conclusion.

Chapter 2

Background

This chapter gives the necessary background information regarding image processing. Section 2.1 describes the process of image formation and degradation and Section 2.2 presents a way to reverse the degradation process with an algorithm called Richardson-Lucy deconvolution. The main image quality metrics are presented in Section 2.3. A summary of the state-of-the-art Richardson-Lucy deconvolution hardware implementations are given in Section 2.4, where a brief overview of a specialization project is given in Section 2.4.2.

2.1 Image Degradation

The sensor system optics consists of several components, such as lenses, mirrors, beam splitters, etc. The optics collect the upcoming light from the *object plane* and forms an image in an *image plane*. The transition between the two through the optical system is described by the HIS's *response function*, which models the introduced distortions. Generally, if the light reflected by the surface is spatially *incoherent*, meaning that the phase and the amplitude of the light wave fluctuate randomly, then the image formation can mathematically be written as

$$f(s) = \int H(s, s')g(s')ds' \tag{2.1}$$

where $H(s, s')$ is the instrument's response function, $g(s)$ is the intensity of the object and position s, and $f(s)$ is the intensity measurement of that object formed by the instrument [20]. The variable s has dimensions appropriate to the application, e.g, s is equal to (x, y, λ) for a 3-D hyperspectral data cube. If an

imaging system, in addition to being spatially incoherent, is also assumed to be *isoplanatic*, meaning that the response function is space-invariant, equation (2.1) can be written as a linear shift-invariant system,

$$f(s) = \int H(s - s')g(s')ds'. \tag{2.2}$$

Symbolically, the equation (2.2) can be written as

$$f(s) = H(s) \circledast g(s) \tag{2.3}$$

where $\circledast$ denotes convolution. The $H(s)$ is also called the *Point Spread Function* (PSF) of the optical system [20] or the convolution *kernel*. The terms PSF and kernel are used interchangeably. It is assumed that the neighbouring spectral bands do not affect each other and can be modeled independently. This means that the 3-D hyperspectral data cube can be seen as a composite of P separate 2-D images, where P is the number of spectral bands with each band having $N \times M$ samples. Following the model presented in [21], the 2-D observation model for the hyperspectral image, within a given spectral band, p, is given by

$$\mathbf{Y}^p = \mathbf{H}_D^p \circledast \mathbf{X}^p + \mathbf{N}^p. \tag{2.4}$$

where $\mathbf{Y}^p \in \mathbb{R}^{N \times M}$ is the p-th band of the observed hyperspectral data cube, $\mathbf{X}^p \in \mathbb{R}^{N \times M}$ is the corresponding ideal band, $\mathbf{H}_D^p \in \mathbb{R}^{W_x \times W_y}$ is the degradation kernel and $\mathbf{N}^p \in \mathbb{R}^{N \times M}$ is a signal-independent Gaussian noise [21].

2.1.1 Image Filtering

In the discrete-space form the integral in equation (2.4), is written as a sum. One output sample at the position (i, j) is found as

$$Y^p(i,j) = \sum_{n=i-W_y/2}^{i+W_y/2} \sum_{m=j-W_x/2}^{j+W_x/2} H_D^p(i-n, j-n)X^p(n,m) + N^p(i,j). \tag{2.5}$$

If the size of the kernel is an odd number, $W/2$ is rounded down [22]. The spatial filtering of a whole input image happens by moving the kernel and processing, using the equation (2.5), the input samples spanned by the kernel. A visual illustration of the convolution process, assuming the absence of the additive

noise, is shown in Figure 2.1, where the blue matrix illustrates an input image of size 4×5, the yellow matrix is the kernel of size 3×3 and the green matrix is the resulting output image. The light gray image borders represent the missing output samples.

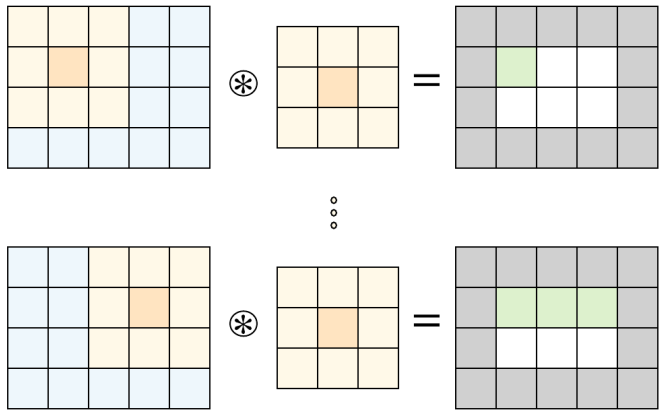


Figure 2.1: Spatial filtering by the means of a moving window.

The following rows can be produced in two ways, either by moving the whole kernel down one row, as shown in Figure 2.2, or by continuing moving the kernel elements column by column, as shown in Figure 2.3.

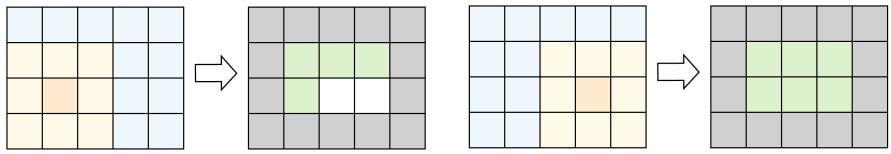


Figure 2.2: The second row produced by moving the whole kernel one sample down.

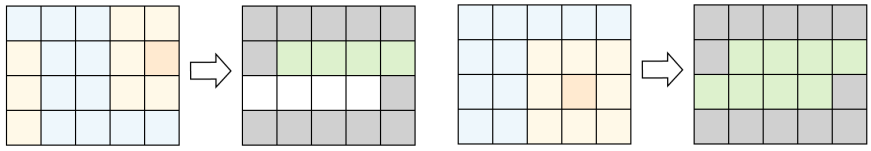


Figure 2.3: The second row produced by moving the kernel column by column.

As seen in Figure 2.2 and Figure 2.3, the output image size depends on the way the kernel is moved. In the case of moving the whole kernel down one row, the linear spatial filtering will output an image of size $(N-(W_x-1), M-(W_y-1))$, i.e., in this case the output image is equal to 2×3, as illustrated by the green matrix in Figure 2.2. Border elements, marked gray, are missing in this scenario. In the case of moving the kernel column by column the output image width will be partially preserved and the height will be shorten to the size equal to $M-(W_y-1)$. The partial image width preservation can be seen in Figure 2.3, where the output pixels are marked green. It should be noted that the first sample(s) in the first valid output row and the last sample(s) in the last valid output row are missing. The number of the missing samples depends on the kernel size and is equal to $(W-1)/2$, if $W = W_x = W_y$. These missing sample can be estimated by setting their values equal to the first valid neighbour value. A method for estimating the rest of the missing samples will be presented later.

2.1.2 Separability

A 2-D convolution is a computationally intensive operation, with a computational complexity equal to $\mathcal{O}(MNW_xW_y)$. Assuming that the kernel is separable, the computational complexity can be minimized to $\mathcal{O}(MN(W_x+W_y))$, which results in a speed-up by a factor equal to $W_xW_y/(W_x+W_y)$. Separability means that a 2-D kernel can be decomposed into two 1-D kernel. For example, the kernel, $\mathbf{H}$, of size 3×3 used in Figure 2.1 can be separated into one vertical kernel, $\mathbf{H}_y$, of size 3×1 and one horizontal kernel, $\mathbf{H}_x$, of size 1×3, where

$$\mathbf{H} = \mathbf{H}_y \circledast \mathbf{H}_x. \tag{2.6}$$

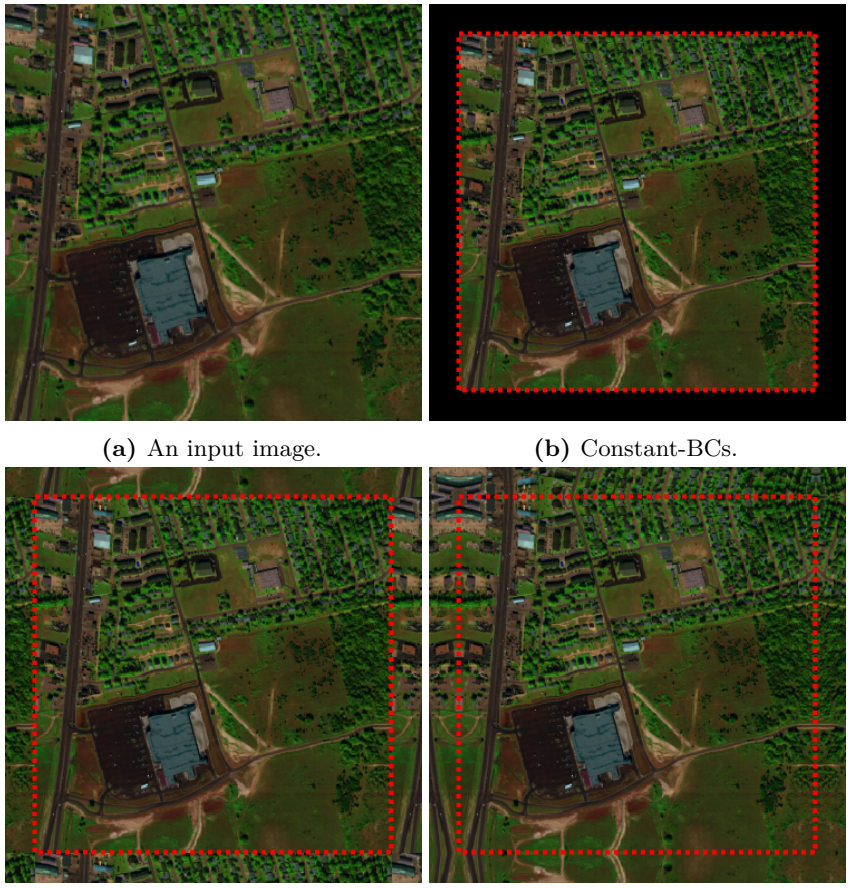
The 2-D separable convolution is computed by first performing the convolution of the input image with the vertical kernel and then convolving the intermediate result with the horizontal kernel.

2.1.3 Border Handling

Convolution, as seen in Figure 2.1, does not preserve the initial image size, thus a pre-processing of data is needed. This can be done by approximating the sample values outside the input image borders. This problem can be solved by imposing a prior *Boundary Conditions* (BCs) on an input image, thus extending an input image with synthetic sample values [23]. The most common BCs are

- *Constant*-BCs, where all samples outside the FOV are assumed to be equal to some constant value, for example zero, as shown in Figure 2.4(b).
- *Periodic*-BCs, assumes the repetition of the object outside the boundaries in all directions, as shown in Figure 2.4(c).
- *Reflective*-BCs, where the samples outside the FOV are a mirrored version of the object inside the FOV, as shown in Figure 2.4(d).

Constant-BCs, where the constant is equal to zero, is the simplest solution, although the least reliable [23]. Assuming periodic-BCs allow the use of the cyclic convolution which can be implemented via *Fast Fourier transform* (FFT) [23] or partially implemented by the convolution method shown in Figure 2.3. The reflective-BCs preserve the boundary continuities. All these three synthetic BCs are designed mostly for the easier computation and are usually the cause of *ringing artifacts*, shown in Figure 2.6(a), at the image borders, as none of them estimate the missing boundary pixels correctly. If the BCs are used in the restoration process with an iterative restoration algorithm, the ringing artifacts tends to worsen, as they will propagate throughout the entire image [24].



(a) An input image. **(b)** Constant-BCs.

(c) Periodic-BCs. **(d)** Reflective-BCs.

Figure 2.4: An example showing different BCs applied on the same input image of size 307×307 shown in (a). Red borders indicate FOV. The extended image is of size 359×359.

2.2 Image Restoration

Image restoration is a well-known image processing problem, where the goal is to find the best approximation of an original, undistorted image from the knowledge of a blurred and noisy observation. If the response function is assumed to be known, the image restoration process is referred to as a *non-blind* restoration, where the only unknown is the undistorted image. Otherwise, if both the PSF and the ideal image are unknown, the image restoration is called *blind*. In this project the non-blind deconvolution for 3-D images is explored.

A general image restoration model is shown in Figure 2.5. It consists of two blocks, the image degradation, described in Section 2.1 and image restoration. An ideal image $f(s)$ is filtered with a degradation function, $h_D(s)$, and further distorted by an additive noise, $\eta(s)$, resulting in a degraded image, $g(s)$. This degraded image is then passed through a restoration filter $h_R(s)$ to produce a restored image $\hat{f}(s)$. The restoration filter depends on the chosen restoration algorithm and the variable s depends on the data dimensions.

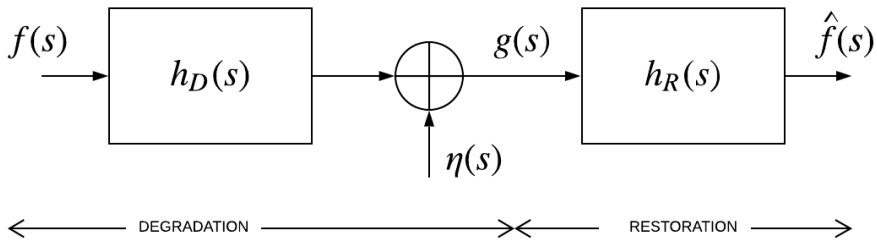


Figure 2.5: A general block diagram model representing the degradation and restoration of the acquired data.

Classical 2-D restoration algorithms such as Wiener filtering [25] and Constrained Least Squares filtering [26] have been also developed to solve 3-D problems. The method in [27] assumes the separability between the spectral and spatial domain. The restoration using Wiener filtering is done independently for each individual 2-D band of the 3-D data. It is stated that no cross-channel, where channel refers to a spectral band, information is lost during the restoration process. The slow execution time is the downside of the described algorithm, as p individual models from equation (2.4) need to be solved. In [28], a modified Wiener filtering is performed without using the separability assumption. It is stated that restoration using the channel-dependent model is better in term of the mean square error than the restoration using a channel-independent model. An alternative to the proposed algorithms in [28] and [27] is presented in [29], where a multichannel Least Squares filtering is presented. It is stated that in the presence of strong cross-channel correlations, the multichannel model outperforms the single-channel

models. The drawback of the described solutions is their inefficiency to restore the frequencies beyond the PSF bandwidth [21]. This results in ringing artifacts, which occurs due to the generation of the negative-value pixels in the deconvolved image. It should be noted that a pixel intensity value can generally be viewed as the number of photons or electrons hitting an imaging detector, and this number cannot be negative, thus a positivity constraint should be used. In [21], it is mentioned that in case of Poisson noise the Richardson-Lucy algorithm can be used. It is a non-linear and iterative algorithm and as claimed in [30] it can achieve both image restoration and super-resolution, where super-resolution refers to the restoration of the high-frequency components.

2.2.1 Richardson-Lucy algorithm

An iterative algorithm developed independently by Richardson [31] and Lucy [32], henceforth called *RL-deconvolution*, is a well-known deconvolution algorithm, which has been applied in a various applications, ranging from astronomy to microscopy. The algorithm has also been derived using a maximum likelihood model for Poisson statistics by Shepp and Vardi [33], where the pixel intensity at a position (i, j) is assumed to be a random variable following the Poisson distribution [34]. If the RL-deconvolution is done on the 2-D data, the RL-deconvolution algorithm is given by [30]

$$\hat{\mathbf{X}}^p_{(k+1)} = \hat{\mathbf{X}}^p_{(k)} \Big[\frac{\mathbf{Y}^p}{\mathbf{H}^p_R \circledast \hat{\mathbf{X}}^p_{(k)}} \odot \mathbf{H}^p_R \Big] \tag{2.7}$$

where $p = 1, \ldots, P$ is the p-th spectral band of the 3-D hyperspectral data cube, shown in Figure 1.4, $\hat{\mathbf{X}}^p_{(k)}$ is the estimate of $\mathbf{X}^p$ after k iterations, $\odot$ is the correlation operator, $\circledast$ is the convolution operator, $\mathbf{Y}^p$ is the p-th band of the acquired hyperspectral image and $\mathbf{H}^p_R$ is the restoration kernel. The positivity constraint holds as long as the initial estimate value $\hat{\mathbf{X}}^p_{(0)}$ is positive, on the other hand, if any of the components becomes equal to zero, it will remain zero for all k [34]. In [34], the initial estimate value is set to the mean value of the input image

$$\hat{\mathbf{X}}^p_{(0)} = \frac{\sum^N \sum^M \mathbf{Y}^p}{N \times M} \tag{2.8}$$

where $N \times M$ is the total number of pixels in one spectral band, but it can also be set to any other positive value.

The RL-deconvolution converges slowly [35], where the pixel value changes most rapidly in the first few iterations, and then slows down as the restoration

progresses. In the case when the additive noise is present, it can be necessary to terminate the restoration process earlier to prevent the possible noise amplification. The RL-deconvolution algorithm preserves the non-negativity in the images [30] and is also robust against small errors in kernel estimations [20]. As mentioned in [30], super-resolution can be achieved due to the non-linearity of the function and the positivity constraint. It should also be mentioned that even though the algorithm is derived assuming the Poisson distributed noise, it is shown in [34], that within certain limits the restoration could also be performed on the data corrupted by the Gaussian noise.

2.2.2 Acceleration of RL-deconvolution

Due to slow convergence, a way to accelerate the RL-deconvolution is needed. A simple method, refereed to as a "multiplicative relaxation" [36] is introduced in [37], where the use of an additional parameter, β, is suggested. The equation (2.7) can be rewritten as

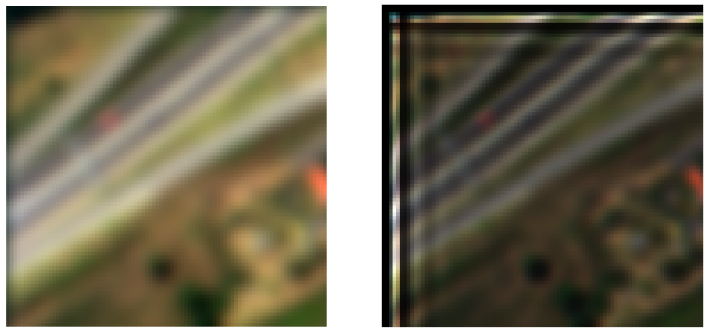
$$\hat{\mathbf{X}}^p_{(k+1)} = \hat{\mathbf{X}}^p_{(k)} \Big[\frac{\mathbf{Y}^p}{\mathbf{H}^p_R \circledast \hat{\mathbf{X}}^p_{(k)}} \odot \mathbf{H}^p_R \Big]^\beta \tag{2.9}$$

where $\beta > 1$ is the exponential correction factor [35]. As stated in [37], the convergence rate is improved. The possible drawback is the lack of stability in the convergence. In [34], it is proved that the number of iterations is reduced by a factor of β, and that β should not be bigger than 2, otherwise the solution starts diverging. It should be noted that for $\beta = 2$, the overall computational cost of one iteration does not increase significantly, therefore when a large number of iterations are needed this simple acceleration approach can be incorporated.

2.2.3 Image Restoration Artifacts

Section 2.2 mentions the ringing artifacts caused by the linear deconvolution. This problem is also seen in the nonlinear algorithms, such as RL-deconvolution. There are several possible causes of the artifacts. As mentioned in Subsection 2.1.3, the unwanted ringing artifacts can arise at the image borders, which are explained as follows: in order for the convolution to produce an output of the same size as the input, the samples outside the optics's FOV are needed. If an object is completely inside the FOV, and the assumptions of the rest of the surroundings can be made, then the restoration without the artifacts is possible. Otherwise, if the samples outside the FOV are unknown and cannot be approximated, then the deconvolved image is going to suffer from ringing artifacts, caused by the the boundary discontinuities [38]. Different boundary conditions cause a different

amount of distortion. For example if Zero-BCs are used on a degraded image shown in Figure 2.6(a), the boundaries of the restored image are going to have ripples as shown in Figure 2.6(b).

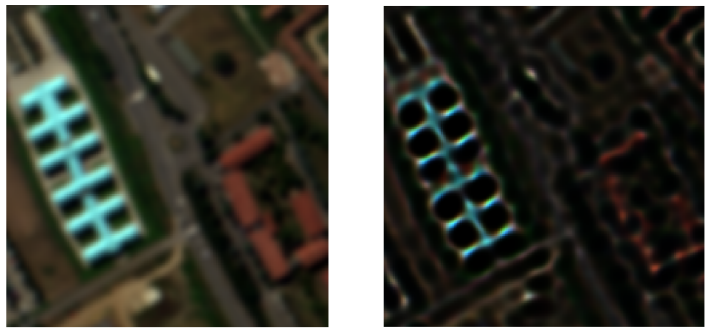


(a) Degraded image.

(b) Boundary ripple effects on the restored image.

Figure 2.6: The restoration effects when using the Zero-BCs on the degraded image.

The ringing artifact can also occur around the edges and bright point sources. In [39], the cause of the latter distortion is contributed to the use of the wrong restoration kernel. Figure 2.7(a) shows an image degraded with a Gaussian kernel with $\sigma = 2.3$ and the size equal to 7×7. Figure 2.7(b) shows the restoration result when using a wrong restoration kernel, in this case equal to a Gaussian kernel with $\sigma = 2.6$ and the size equal to 9×9.



(a) **(b)**

Figure 2.7: The restoration effects on image degraded with Gaussian PSF with $\sigma = 2.3$ (a), which is restored with a restoration kernel not equal to the degradation kernel (b).

The presence of additive noise can also be a cause of the artifacts. In order to prevent the ringing artifacts, one can make use of the regularization techniques

[39], or simply terminate the iterative algorithm earlier.

2.3 Image Quality Assessment

For simulation purposes only, the relative reconstruction error, *RRE*, can be used to test how similar the reconstructed image $\hat{\mathbf{X}}_k^p$ when compared with the ideal image $\mathbf{X}_{true}^p$

$$RRE = \frac{||\hat{\mathbf{X}}_k^p - \mathbf{X}_{true}^p||}{||\mathbf{X}_{true}^p||} \tag{2.10}$$

where ||.|| denotes the Euclidean distance.

The performance of the RL-deconvolution algorithm can also be evaluated using the *Peak Signal-to-Noise Ratio* (PSNR) as it is the most used image quality assessment measurement. The PSNR is calculated band-by-band and is mathematically expressed as

$$PSNR = 10log_{10}\frac{(M \times N)(\mathbf{X}_{true(max)}^p)^2}{||\mathbf{X}_{true}^p - \hat{\mathbf{X}}_k^p||^2} \tag{2.11}$$

where $\mathbf{X}_{true(max)}^p$ is the maximum value of the image $\mathbf{X}_{true}^p$ and the $M \times N$ is the number of samples in one band [40]. The denominator in equation (2.11) refers to the *Mean Square Error* (MSE). Ideally, the error between the restored image and the ideal image is zero, which results in MSE = 0, which in turn results in PSNR being equal to infinity. Thus, the higher PSNR value is, the better is the restoration of the image.

PSNR performs poorly compared to the human visual system and properties such as luminance, contrast or structure in the images are ignored. *Structural Similarity Index* (SSIM) takes these parameters into the account and measures similarity between a restored image and the reference image in a way which is more similar to the human eye.

$$SSIM(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p) = [l(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p)^\alpha \cdot [c(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p)]^\beta \cdot [s(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p)]^\gamma \tag{2.12}$$

where $l(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p)$ is related to the luminance difference, $c(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p)$ to the contrast differences, and $s(\hat{\mathbf{X}}_k^p, \mathbf{X}_{true}^p)$ to the structure variations, where all three

parameters are calculated between $\hat{\mathbf{X}}_k^p$ and $\mathbf{X}_{true}^p$. The parameters α, β and γ defines the relative importance of each component. SSIM lies between 0 and 1, where 1 refers to the complete similarity between the restored image an the reference image [40].

The equations (2.11) and (2.12) are computed for one 2-D band only. In order to evaluate the quality of the whole 3-D hyperspectral cube, a *mean PSNR* (M-PSNR) and a *mean of SSIM* (M-SSIM) are found by first computing the PSNR and SSIM band-by-band and then averaging the results [40]. The M-SSIM computations are done using built-in MATLAB R2018b function.

2.4 State-of-The-Art

In Section 2.4.1, the state-of-the-art software approach for 3-D RL-deconvolution is presented. In Section 2.4.2, Hardware/Software codesign architecture of the RL-deconvolution done during the specialization project in autumn 2018 at NTNU is reviewed. Finally in Section 2.4.3, the state-of-the-art hardware implementations of RL-deconvolution are presented.

2.4.1 3-D Richarsdon-Lucy Deconvolution

The article [41] presents a method to do a 3-D RL-deconvolution on the hyperspectral images acquired with a *Short Wave Infrared* (SWIR), pushbroom imaging system. The method enhances the spatial resolution in both the cross- and in-track directions simultaneously with the cross-channel displacement corrections. The spectral image o is modeled as a 3-D convolution between an undistorted image $g(u, w, z)$ with the spatially-variant response function $h(u, w, z)$,

$$o(u, w, z) = g(u, w, z) * h(u, w, z) + b(u, w, z) + \epsilon(u, w, z) \tag{2.13}$$

where u, w and z are the cross-track, spectral and in-track directions, respectively, variable b accounts for the temperature variations and ϵ stands for the difference between the acquired image and the modeled one. The authors state that the reduction of displacement and blur can be done simultaneously, when the response function is known, using the 3-D RL-deconvolution. At the end of each iteration, a standard deviation of the residual, $r(u, w, z)^{(k)}$, is compared to the a standard deviation of the previous residual, $r(u, w, z)^{(k-1)}$, and the algorithm is stopped if a pre-defined threshold is met. The tests are done on the images of size $320 \times 235 \times Z$ pixels, where Z is the extend in the in-track direction. It is stated that the spatial resolution is enhanced over all the bands, having the biggest effect on the bands with the lowest acquired resolution. The method is suitable for an offline deconvolution, when a whole hyperspectral data cube is available. The paper does not state how the image boundaries are treated nor what the execution time is.

2.4.2 Specialization project overview

The work of this thesis is based on the research performed during a specialization project [42] conducted during the autumn of 2018 at NTNU. In the specialization project, the RL-deconvolution was studied by first writing a C-script and comparing the results to the results generated by the built-in MATLAB function, *deconvlucy(IMG, PSF, ITER)* and then implementing the parts of algorithm in hardware, resulting in a HW/SW codesign.

A 3-D hyperspectral data cube is assumed to be cross-channel independent, therefore each 2-D band of size $N \times M$ can be processed independently. Let $\mathbf{Y}^p$ denote the p-th band from the acquired and degraded hyperspectral data cube, $\hat{\mathbf{X}}^p{}_{(k)}$ denote the estimated, underlying ideal p-th band after k iterations and $\mathbf{H}^p_R$ denote the restoration kernel for the p-th band. The main steps of the standard RL-deconvolution from equation (2.7) are summarized as follows:

- decide an initial estimate, $\hat{\mathbf{X}}^p{}_{(0)}$
- compute the residual, $residual = \dfrac{\mathbf{Y}^p}{\mathbf{H}^p_R \circledast \hat{\mathbf{X}}^p_{(k)}} = \dfrac{\mathbf{Y}^p}{\mathbf{Y'}^p_{(k)}}$
- calculate the correction factor, $\phi_{(k)} = residual \odot \mathbf{H}^p_R$
- lastly, update the initial estimate, $\hat{\mathbf{X}}^p{}_{(k+1)} = \hat{\mathbf{X}}^p{}_{(k)} \times \phi_{(k)}$.

where $\circledast$ denotes the 2-D convolution and $\odot$ denotes 2-D cross-correlation. If the restoration kernel is symmetric, the cross-correlation is the same as the convolution. A block diagram illustrating one RL-deconvolution iteration for one band is shown in Figure 2.8.

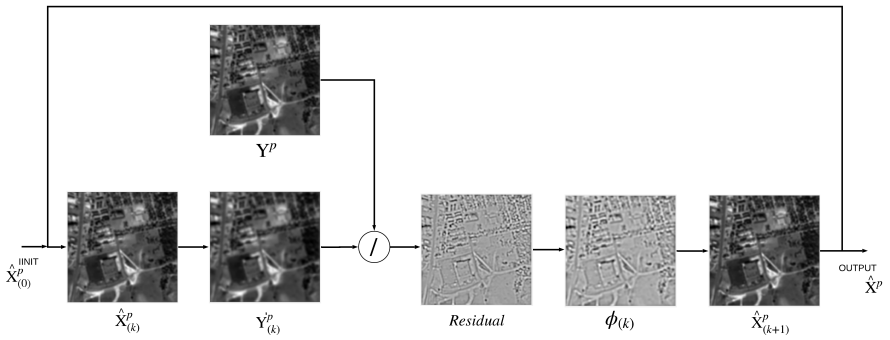
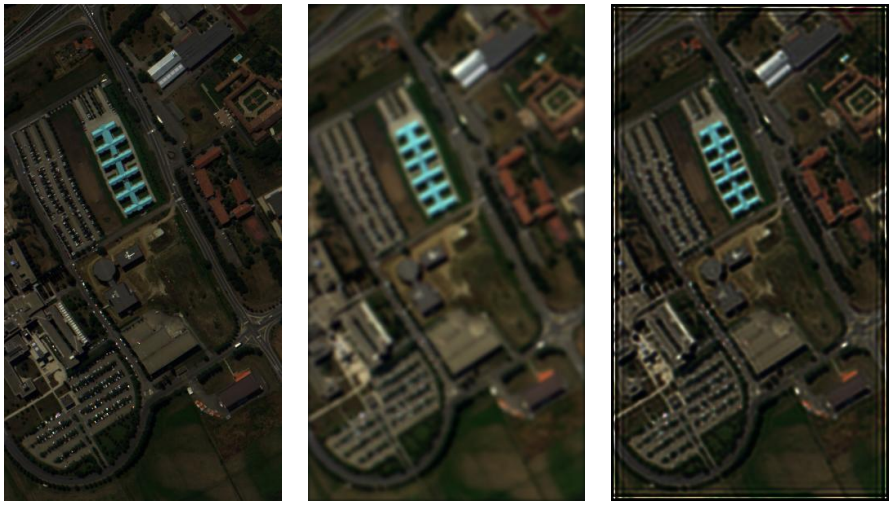


Figure 2.8: An illustration of one iteration of RL-deconvolution algorithm.

The algorithm was tested on the several publicly available hyperspectral data sets, such as Pavia university data set [43] and the Urban data set [4]. The images were blurred by the Gaussian blur and by the additive Gaussian noise prior to the deconvolution. For example, the image in Figure 2.9(a) is the original three-color composite made of spectral bands (50, 30, 10) is used as a reference. This reference image is degraded with Gaussian kernel with $\sigma = 2.9$, shown in Figure 2.9(b). The RL-deconvolution is run for 50 iterations and the deconvolved result is shown in Figure 2.9(c). The relative improvement in M-PSNR and M-SSIM compared to the reference image is equal to 2.6 dB and 15.37 %, respectively. M-PSNR and M-SSIM are found for a cropped image, without taking borders into the account, this is due to the zero-BCs used to approximated the image borders in this implementation.



(a) **(b)** **(c)**

Figure 2.9: The reference image (a) degraded with a Gaussian kernel with $\sigma = 2.9$ (b) is deblurred using 50 RL-deconvolution iterations (c).

In order to accelerate the software-based approach, a hardware accelerator was designed specifically for the 2-D convolution computations. The RL-deconvolution algorithm was implemented as a Hardware/Software codesign. The implementation was tested on the ZedBoard development board with the ARM Cortex-A9 processor and Zynq-7020 FPGA. The block diagram for Hardware/Software codesign architecture of the RL-deconvolution is shown in Figure 2.10. The accelerator is connected to the processing system through the AXI *Direct Memory Access* (DMA) module. The CONCAT module, shown in Figure 2.10, is responsible for correctly invoking the processor. The block diagram of the convolution accelerator is shown in Figure 2.11. The accelerator consists of a convolution module and write/read controllers.

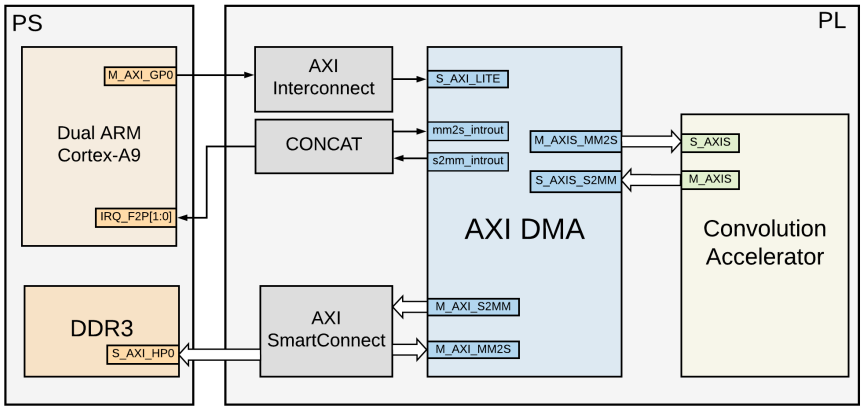


Figure 2.10: Block diagram for the Hardware/Software codesign implementation of the RL-deconvolution algorithm.

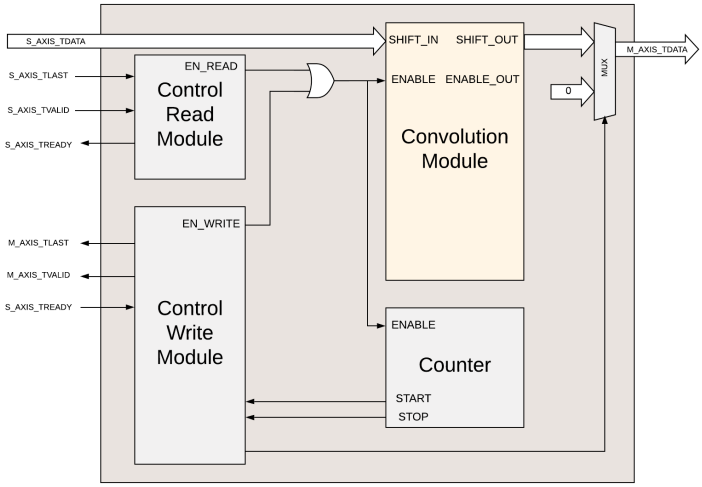


Figure 2.11: Block diagram of the convolution accelerator.

The utilization report when using an image of size 640×310 is shown in Table 2.1. Hardware/Software codesign was compared to the Software-Only implementation on the same platform and the achieved speed-up was equal to $2.7\times$.

Table 2.1: Utilization summary after the implementation for the input image of size 640×310 with address width 16-bits.

Resource	Utilization	Available	Utilization %
LUT	5054	53200	9.50
LUTRAM	601	17400	3.45
FF	6131	106400	5.76
BRAM	6	140	4.29

2.4.3 State-of-The-Art hardware implementations of RL-Deconvolution algorithm

There are a few different hardware implementations of 2-D RL-deconvolution algorithm. Some assume a space-invariant PSF [44] [45] [46], while there are also implementations with a space-variant PSF [47]. Authors in [44] assume a shift-invariant PSF, meaning that a whole scene is affected by the same degradation kernel. The RL-deconvolution is performed in frequency domain. The presented architecture is *Digital Signal Processor* (DSP) based, and uses Virtex-4 FPGA as a co-processor. The maximum processing frequency is 100 MHz. The proposed system executes the algorithm on images for of size 64×65. In [47], the PSF is assumed to be shift-variant. To ease the computational complexity and memory requirements, PSFs are described by sparse matrices. The proposed architecture is implemented on Altera Stratix V. Deconvolution is tested assuming both the motion blur and the lens distortion, modeled with the Gaussian PSF. Every sample in an image is associated with a unique PSF. The maximum processing frequency is not stated. The implementation is tested on an image of size 640×480. Authors in [] implement the accelerated RL-deconvolution, with β values among 1 to 3 at the first iteration. The kernel is assumed to be space-invariant and separable, and a 2-D convolution is decomposed into two 1-D convolutions. The solution implements a fixed amount of RL-deconvolution iterations, equal to 2 iterations. The algorithm is implemented as a one continuous datapath, which simplifies the control system. The implementation uses a kernel size equal to 11×11 and images of size 640×480. The algorithm is implemented on a Xilinx Virtex 3 XC2VP50, the maximum frequency is 63 MHz and an achieved throughput is equal to 60 MP/s (megapixels per second). Finally, the authors in [45] present a fully ported hardware implementation of the RL-deconvolution algorithm. The presented architecture is scalable from 3×3 to 9×9 kernel sizes. Similar to the [46], the architecture is implemented to process a fixed amount of iterations, in this case equal to 10. The presented architecture is tested on Stratix V device. Maximum processing frequency is equal to 61 MHz. This is the only implementation presenting a way to keep the original image size.

Chapter 3

Analysis of Richardson-Lucy Deconvolution Algorithm

The main chapter objectives are to test some of the assumptions made in Chapter 2 and give a solid ground for designing a hardware architecture. It is stated by several authors that the RL-deconvolution algorithm is robust against small changes in the restoration kernel, thus the goal of the Section 3.3.1 is to observe how these errors in kernel estimation affects the restoration results. In Section 3.3.2, the effects of different types of boundary conditions are examined. In Section 3.3.3, the conversion from the floating-point to the fixed-point representation is looked at. Finally, Section 3.4 examines how the presence of noise affects the restoration results.

3.1 Hyperspectral Data Sets

Two different hyperspectral data sets are used in the RL-deconvolution algorithm analysis, both are taken by different sensors (i.e., *Hyperspectral Digital Imagery Collection Experiment* (HYDICE) and HICO) and have a varying amount of detail.

The *Urban* hyperspectral data collected by the HYDICE sensor has a total of $P = 162$ spectral bands, each of size 307×307, where each sample corresponds to a 2×2 m^2 area [4]. The three-band composite (100, 55, 30) taken from the Urban data set is shown in Figure 3.1.

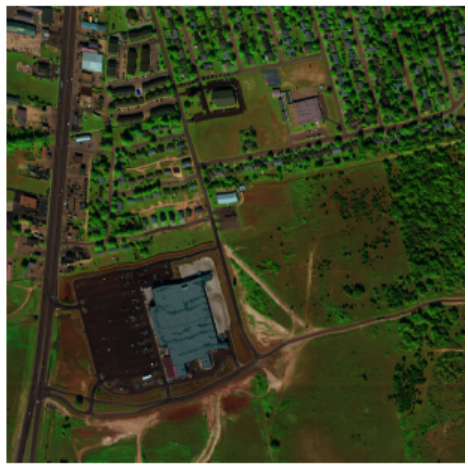


Figure 3.1: A three-color composite of three spectral bands (100, 55, 30) from Urban data set [4].

The HICO spectrometer is designed specifically for the ocean observations with spatial resolution equal to 90 m. The hyperspectral data is of size $512 \times 2000 \times 128$ [2]. The data used here represents the coast near Christchurch, New Zealand, called *Coast*. The three-band composite (18, 25, 53) is shown in Figure 3.2.

Figure 3.2: Image to the left shows a three-color composite of three spectral bands (18, 25, 53) from HICO data set [2].

3.2 Restoration and Degradation Kernel

For simulation purposes, both the degradation kernel, $\mathbf{H}_D^p$, and the restoration kernel, $\mathbf{H}_R^p$, are set to be equal to the Gaussian PSF, which is used to model e.g., the optical blur. The 2-D Gaussian function is given by

$$PSF(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{\sigma^2}\right) \tag{3.1}$$

where the standard deviation, σ, determines the width of the PSF and controls the amount of blur in the image. The Gaussian function is separable, thus the equation (2.6) holds and the equation (3.1) can be written as

$$PSF = PSF_x \circledast PSF_y = \left[\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{x^2}{2\sigma^2}\right)\right] \circledast \left[\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{y^2}{2\sigma^2}\right)\right] \tag{3.2}$$

Each 2-D image, $\mathbf{X}^p$, in the hyperspectral data cube is degraded and restored with the same Gaussian degradation/restoration kernel thus the superscript p in $\mathbf{H}_R^p$ and $\mathbf{H}_D^p$ is not needed.

3.2.1 Kernel Size and Quantization

The designed hardware architecture uses fixed-point data representation, thus an appropriate quantization of the kernel coefficients is needed. The size of the Gaussian kernel depends on the chosen σ, where larger σs results in larger kernels. Generally, values beyond three standard deviations from the mean value are considered to the equal to zero. So in order to keep all the non-zero kernel values, the minimum kernel size should be equal to $minimum = 3\sigma$. Assuming a 1-D kernel with $\sigma = 2.3$ is used, then the minimum theoretical size of the kernel is equal to $3 \times 2.3 = 6.9 = 7$. The normalised kernel coefficients in a floating-point representation with four significant digits after the radix point are equal to

$$h_{float} = \begin{Bmatrix} 0.0847 & 0.1358 & 0.1804 & 0.1982 & 0.1804 & 0.1358 & 0.0847 \end{Bmatrix} \tag{3.3}$$

The fixed-point representation is found by

$$h_{fixed} = round(h_{float} \times 2^F) \tag{3.4}$$

$$h'_{float} = h_{fixed} \times 2^{-F} \tag{3.5}$$

where h'_{float} is the approximation of the original floating-point number h_{float} and F is the number of fractional bits. The kernel coefficients have all values smaller than 1. Therefore, the fractional length can be set to be equal to the wordlength and the value of the wordlength is decided experimentally by calculating the Euclidean distance between the floating point, h_{float}, and the approximation of the floating point, h'_{float}, the plot of the error as a function of fractional bit length is shown in Figure 3.3. As seen in Figure 3.3, the more bits are used in the quantization, the smaller is the error. If fractional bit length is set to be equal or bigger than 7 bits, the resulting error is smaller than 0.5 %. By choosing $F = 7$ bits, the kernel coefficients can be stored in an 8-bit integer. The kernel coefficients in a fixed-point representation with $F = 7$ bits are equal to

$$h_{fixed} = \begin{Bmatrix} 11 & 17 & 23 & 25 & 23 & 17 & 11 \end{Bmatrix}. \tag{3.6}$$

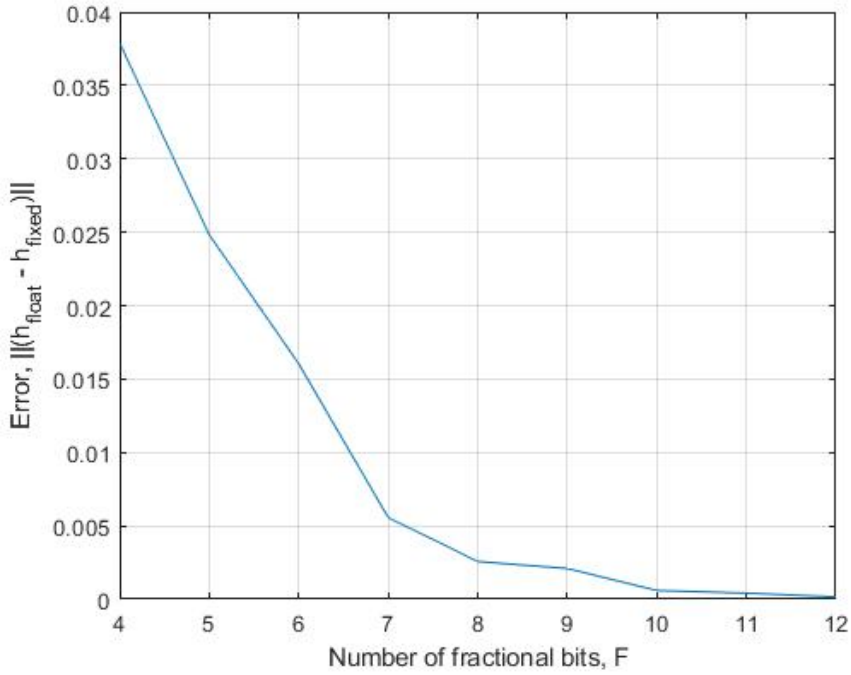
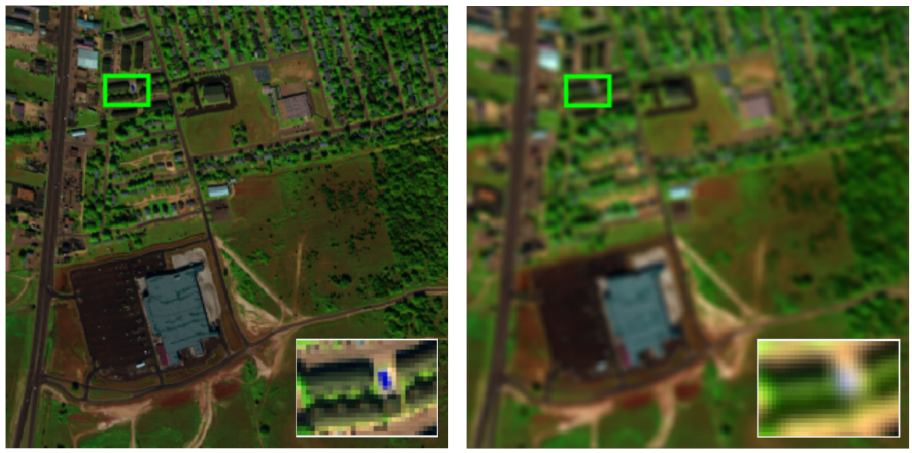


Figure 3.3: Euclidean distance between the floating point, h_{float}, and the approximation of the floating point, h'_{float} as a function of fractional length of the kernel coefficients.

3.3 Hyperspectral Data - Urban

In this section, three spectral bands from the Urban data set are used. The chosen bands are closed to the red-, green- and blue-wavelengths corresponding to the bands 30, 55 and 100. The degradation-free image, used as a reference, is shown in Figure 3.4(a). The image is synthetically degraded by the Gaussian blur with zero mean and $\sigma = 2.3$, shown in Figure 3.4(b). All the individual bands are degraded independently. The M-PSNR and the M-SSIM between the blurred image and the reference image are equal to 41.15 dB and 0.7859, respectively. In order to preserve the output image size, the input images are re-sized before each convolution for each iteration of RL-deconvolution.



(a) **(b)**

Figure 3.4: Original three-color composite of three spectral bands (100, 55, 30) from Urban data set (a) is degraded with the Gaussian blur with $\sigma = 2.3$ and size 7×7. The small rectangular images on the right corners shows a close-up of an area marked with a green rectangular.

3.3.1 Varying Standard Deviation

The effects of the errors in kernel estimation are shown in this section. In addition, the standard RL-deconvolution algorithm given by equation (2.7), is compared to the accelerated RL-deconvolution algorithm in equation (2.9). The degraded image is restored with the Gaussian kernel having $\sigma = [1.0, 2.0, 2.3, 2.6, 5.0]$, which results in the reconstruction kernel both smaller, equal and bigger than the degradation kernel. Each kernel size is set to be equal to 3σ. The resulting mean reconstruction error plotted as a function of the number of iterations is shown in Figure 3.5, where the solid lines denote the standard RL-deconvolution and the dashed lines denote the accelerated RL-deconvolution algorithm with $\beta = 2$.

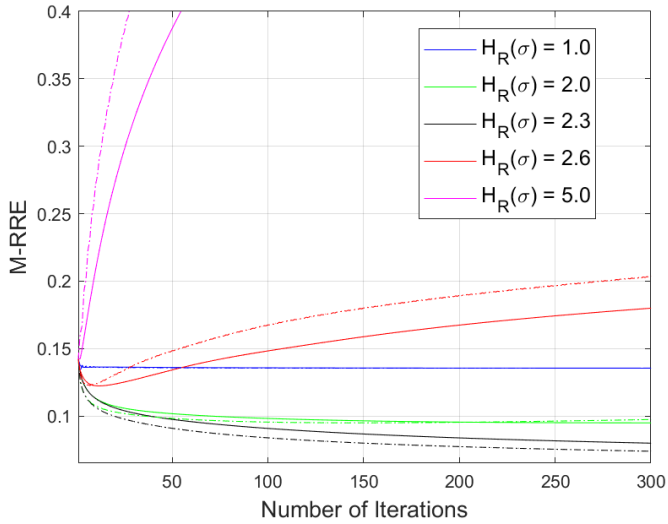


Figure 3.5: M-RRE as a function of the number of iterations for different reconstruction kernels using standard RL-deconvolution (solid lines) and accelerated RL-deconvolution (dashed lines). Plot does not show M-RRE bigger than 4 %.

The minimum M-RRE is found for each reconstruction kernel. The M-PSNR and the M-SSIM values at the calculated minimum are shown in Table 3.1 and Table 3.2, for the standard RL-algorithm and the accelerated RL-algorithm respectively. From Table 3.1 and Table 3.2, it is seen that the accelerated version of the RL-deconvolution algorithm with $\beta = 2$ reaches minimum approximately twice as fast as the standard version, demonstrating that the stated assumption in Chapter 2 holds. This improvement can be significant, when a large number of iterations is needed.

The reconstruction result depends on how good the knowledge about the degradation kernel is, as seen in both the Figure 3.5, and the Table 3.1 and Table 3.2. If restoration kernel is equal to the degradation kernel (i.e., in this example σ

Table 3.1: M-PSNR and M-SSIM at the minimum M-RRE for different kernels running the standard RL-deconvolution.

		M-PSNR(dB)		M-SSIM	
$\mathbf{H}_R(\sigma)$	iter.	M-PSNR	Relative improvement (dB)	M-SSIM	Relative improvement (%)
1.0	300	41.57	0.42	0.8025	2.12
2.0	294	44.61	3.46	0.8985	14.33
2.3	300	46.11	5.97	0.9207	17.16
2.6	22	42.42	1.27	0.8287	5.45
5.0	1	41.14	0.0	0.7888	0.37

Table 3.2: M-PSNR and M-SSIM at the minimum M-RRE for different kernels running the accelerated RL-deconvolution.

		M-PSNR(dB)		M-SSIM	
$\mathbf{H}_R(\sigma)$	iter.	M-PSNR	Relative improvement (dB)	M-SSIM	Relative improvement (%)
1.0	165	41.57	0.42	0.8027	2.14
2.0	147	44.61	3.64	0.8986	14.34
2.3	300	46.79	5.64	0.9361	19.11
2.6	7	42.40	1.26	0.8293	5.52
5.0	1	39.85	-1.29	0.7807	-0.78

= 2.3) the RL-deconvolution algorithm will at one point converge to a suitable solution. Figure 3.6 shows the M-RRE between the image reconstructed with kernel with $\sigma = 2.3$ an the reference image for 5000 RL-deconvolution iterations. The algorithm converges most rapidly for the first ∼500 iterations, and then slows down. The achieved M-PSNR and M-SSIM after 5000 accelerated RL-deconvolution iterations are equal to 48.84 dB and 0.9708, respectively, and the relative improvement in M-PSNR and in M-SSIM after 5000 accelerated RL-deconvolution iterations are equal to 7.69 dB and 23.53 %, respectively. Visual reconstruction results using $\mathbf{H}_R(\sigma) = 2.3$ are shown in Figure 3.8(a) and Figure 3.8(b) for 300 accelerated RL-deconvolution iterations and 5000 iterations, respectively. Although, the M-RRE gets smaller for each iteration, the visual result is similar for $k = 300$ and $k = 5000$.

When the reconstruction kernel is not ideal (i.e., not equal to the degradation kernel) the deconvolution result will depend on how close the restoration kernel's coefficients are to the ideal kernel's coefficients. For example, the restoration kernel $\mathbf{H}_R(\sigma) = 2.6$ is bigger than the degradation kernel used in this example. As seen in Table 3.2 and Figure 3.8(d) the relative improvement in M-SSIM after 22 iterations is equal to 5.52 % and after 300 iterations is equal to -2.38 % for

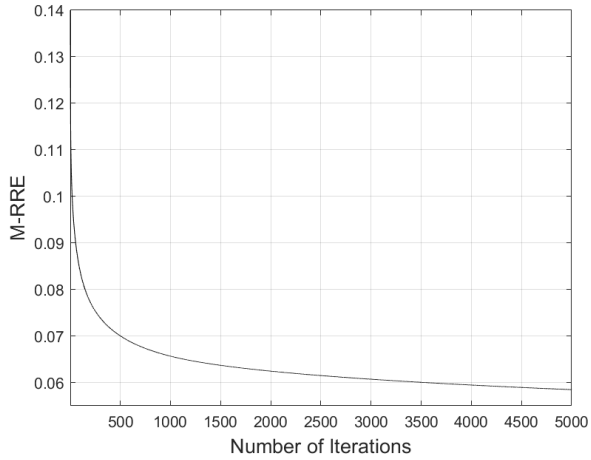
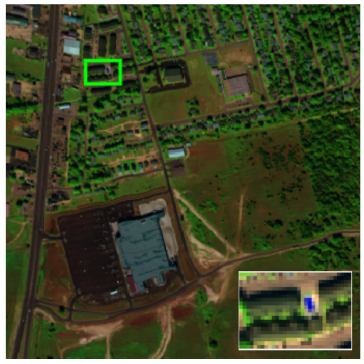


Figure 3.6: Mean reconstruction error as a function of the number of iterations using $\mathbf{H}_R = \mathbf{H}_D$.

the accelerated RL-deconvolution. This means that the restoration using this kernel can improve the spatial image resolution as long as the RL-deconvolution is stopped early. On the other hand, the restoration kernel $\mathbf{H}_R(\sigma) = 5.0$ is too big compared to the degradation kernel. This can be observed in Figure 3.5, where the deconvolution result keeps diverging for each new iteration (the values for M-RRE bigger than 4 % are not shown), and in Table 3.1 and Table 3.2, where the relative improvement in M-SSIM after 1 iteration is equal to 0.37 % for the standard RL-deconvolution and -0.78 % for the accelerated RL-deconvolution. The result is positive, even though not by a significant amount, for the standard RL-deconvolution and negative for the accelerated RL-deconvolution. This would suggest a need for the possibility to perform both algorithms, e.g., in this case the standard RL-deconvolution would be preferred. The restoration results seem to be better when using restoration kernels smaller than the degradation kernel compared to the restoration kernels bigger than the degradation kernel. Table 3.1 and Table 3.2, shows an improvement in M-SSIM for both $\mathbf{H}_R(\sigma) = 1.0$ and $\mathbf{H}_R(\sigma) = 2.0$. As seen in Figure 3.5, the M-RRE is constant, to a greater or lesser extent, with respect to the number of iterations. Hence, RL-deconvolution will give a reasonable result even if not stopped at the minimum reached M-RRE, as shown in Figure 3.7(c), Figure 3.7(d), Figure 3.7(e) and Figure 3.7(f).

Figure 3.7(c), Figure 3.7(e), Figure 3.8(a), Figure 3.8(c) and Figure 3.8(e) shows visual RL-deconvolution results at a minimum M-RRE for kernels with $\sigma = 1.0$, $\sigma = 2.0$, $\sigma = 2.3$, $\sigma = 2.6$ and $\sigma = 5.0$, respectively. Visual RL-deconvolution results in the case when the algorithm is not stopped at the minimum M-RRE are shown in Figure 3.7(d), Figure 3.7(f), Figure 3.8(b), Figure 3.8(d) and Figure 3.8(f) kernels with $\sigma = 1.0$, $\sigma = 2.0$, $\sigma = 2.3$, $\sigma = 2.6$ and $\sigma = 5.0$, respectively.



(a) Reference image.

(b) Degraded with $\mathbf{H}_D(\sigma) = 2.3$. M-PSNR=41.15, M-SSIM=0.7859.

(c) $\mathbf{H}_R(\sigma) = 1.0$, $k = 3$.

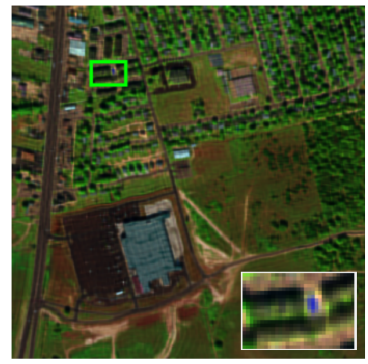
(d) $\mathbf{H}_R(\sigma) = 1.0$, $k = 300$. M-PSNR=41.56, M-SSIM=0.8030.

(e) $\mathbf{H}_R(\sigma) = 2.0$, $k = 147$.

(f) $\mathbf{H}_R(\sigma) = 2.0$, $k = 300$. M-PSNR=44.38, M-SSIM=0.9092.

Figure 3.7: Visual accelerated RL-deconvolution results with varying kernel coefficients. k indicates the number of RL-deconvolution iterations.

(a) $\mathbf{H}_R(\sigma) = 2.3$, $k = 300$.

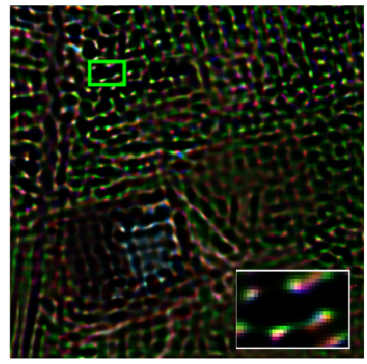


(b) $\mathbf{H}_R(\sigma) = 2.3$, $k = 5000$.
M-PSNR=48.84, M-SSIM=0.9708.

(c) $\mathbf{H}_R(\sigma) = 2.6$, $k = 208$.

(d) $\mathbf{H}_R(\sigma) = 2.6$, $k = 300$.
M-PSNR=38.02, M-SSIM=0.7672

(e) $\mathbf{H}_R(\sigma) = 5.0$, $k = 8$



(f) $\mathbf{H}_R(\sigma) = 5.0$, $k = 300$.
M-PSNR=25.39, M-SSIM=0.2195

Figure 3.8: Visual accelerated RL-deconvolution results with varying kernel coefficients. k indicates the number of RL-deconvolution iterations.

3.3.2 Boundary Conditions

Boundary conditions affect the restored image boundaries. Three types of boundary conditions are tested, Z-BCs, Variable-Constant-BCs (VC-BCs) and a Modified-Periodic-BCs (MP-BCs). It should be noted, that besides the Z-BCs, the other two conditions are named specifically for the project and have not been found in literature. The VC-BCs are the ones used in the simulations above. The constant is set to be equal to the first sample value of the input image and it changes for each iteration depending on the input data. A MP-BCs simulates convolution showed in Figure 2.3, where the image border elements are estimated by "wrapping" kernel around the image. It is modified as it uses the VC-BCs for the upper and lower missing samples of an input image, and Periodic-BCs for the missing samples on the sides of the input image. Here, an input image refers to the data going into the convolution.

The restoration here is done using a Gaussian kernel of size 7×7 with $\sigma = 2.3$. Similarly to the Subsection 3.3.1, both standard- and the accelerated RL-deconvolution algorithms are tested. The M-PSNR and M-SSIM between the degraded data and the reference data for the full frame and cropped frame are shown in Table 3.3.

Table 3.3: M-PSNR and M-SSIM for Full-frame and Cropped-frame compared to the reference data.

	M-PSNR	M-SSIM
Full-frame	41.15	0.7859
Cropped-frame	39.90	0.7802

The RL-deconvolution is run for $k = 100$ iterations. The resulting mean reconstruction error plotted as a function of the number of iterations is shown in Figure 3.9 for both the full frame and the cropped frame. The solid lines correspond to the standard RL-deconvolution and dashed lines correspond to the accelerated RL-deconvolution. M-PSNR and M-SSIM are calculated at the $k = 100$ and the results are shown in Table 3.4 and Table 3.5 for full- and cropped frames respectively. The values are shown for the standard RL-deconvolution algorithm. Visual deconvolution results can be seen in Figure 3.10, where Figure 3.10(a), Figure 3.10(c) and Figure 3.10(e) represent the deconvolution result using Z-BCs, VC-BCs and MP-BCs, respectively. Figure 3.10(b), Figure 3.10(d) and Figure 3.10(f) shows their corresponding cropped frames.

As seen in Figure 3.9 and Figure 3.10, when the boundaries of the deconvolved images are disregarded (i.e., the image borders are cropped) the reconstruction result is similar for all BCs. This can also be seen in Table 3.5, where the differences in both the M-PSNR and M-SSIM are insignificant for the cropped

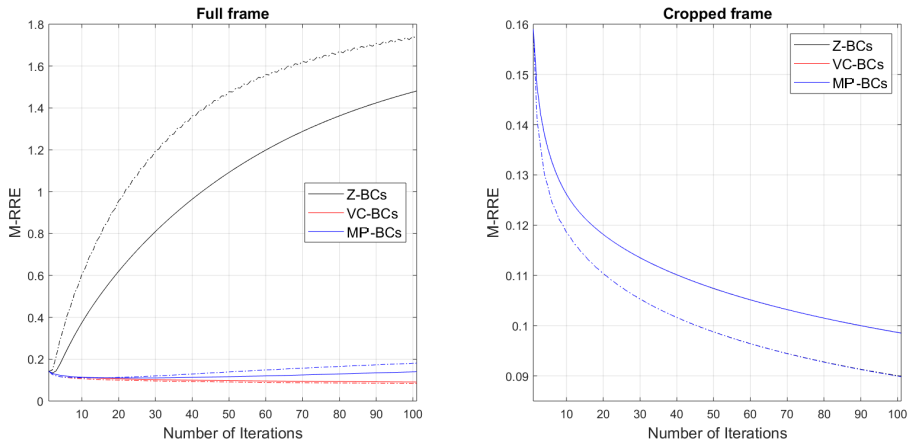


Figure 3.9: Mean reconstruction error as a function of the number of iterations. Deconvolution is done with two Gaussian kernel with $\sigma = 2.3$, and three different BCs, Z-BC, VC-BC and MP-BC.

Table 3.4: M-PSNR and M-SSIM, for image deblurred using a Gaussian kernel of size 7×7 with $\sigma = 2.3$ and three different BCs. Full Frame.

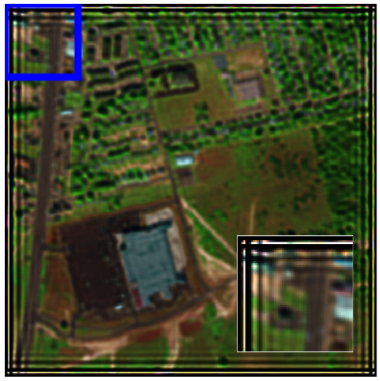
	M-PSNR(dB)		M-SSIM	
BCs	M-PSNR	Relative improvement (dB)	M-SSIM	Relative improvement (%)
Z-BC	20.75	-20.37	0.7805	-0.69
VC-BC	45.00	3.85	0.8920	13.50
MP-BC	41.37	0.23	0.8859	12.72

Table 3.5: M-PSNR and M-SSIM, for image deblurred using a Gaussian kernel of size 7×7 with $\sigma = 2.3$ and three different BCs. Copped Frame.

	M-PSNR(dB)		M-SSIM	
BCs	M-PSNR	Relative improvement (dB)	M-SSIM	Relative improvement (%)
Z-BC	44.01	4.11	0.8855	13.50
VC-BC	44.01	4.11	0.8856	13.51
MP-BC	44.01	4.11	0.8856	13.51

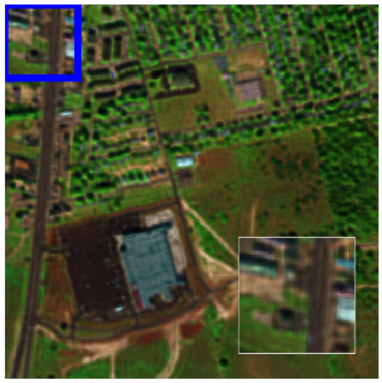
frames in all cases. If the boundaries are included, the deconvolution results differ significantly, where the Z-BCs produces the most visible ringing artifacts around the image borders. The MP-BCs also produces the ringing artifacts on the sides of the output image, although not as visible as the Z-BC. It seems as

the VC-BCs produces the best results both visually as seen in Figure 3.10(c) and from analyzing the quality metrics in Table 3.4.



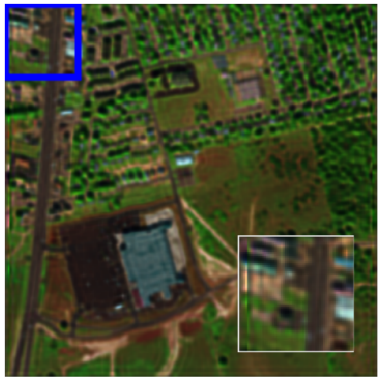
(a) Z-BCs

(b) Z-BCs, close-up



(c) VC-BCs

(d) VC-BCs, close-up



(e) MP-BCs

(f) MP-BCs, close-up.

Figure 3.10: Visual deconvolution results with varying BCs.

3.3.3 Precision Analysis

In described software-based approach, both the input and the output image, and all the intermediate operations are represented in a single precision floating-point format with width equal to 4 bytes. The hardware module for RL-deconvolution is implemented using fixed-point format, therefore a correct conversion from the floating-point format to the fixed-point format is needed. The data precision is tested by deconvolving the degraded image shown in Figure 3.4 using the floating-point C-script and a fixed-point C-script. Firstly, the conversion between floating-point representation and the fixed-point representation method used in the previous work [42] is tested. The described method disregards the fractional part of the data by truncation. It is found that the precision loss gets significant after some number of iterations. In the test case, this number is equal to around 30 iterations, as seen in Figure 3.11. This can be explained as follows: in the first couple of iterations the integer part of the estimated image is dominant and the made changes are noticeable, but after some time the fractional part starts dominating. Since the fractional part is truncated after each operation, the restored image does not improve any further. The achieved *mean squared error* (MSE) between the floating-point and fixed-point data is equal to MSE = 121.07 and the M-SSIM = 0.9814.

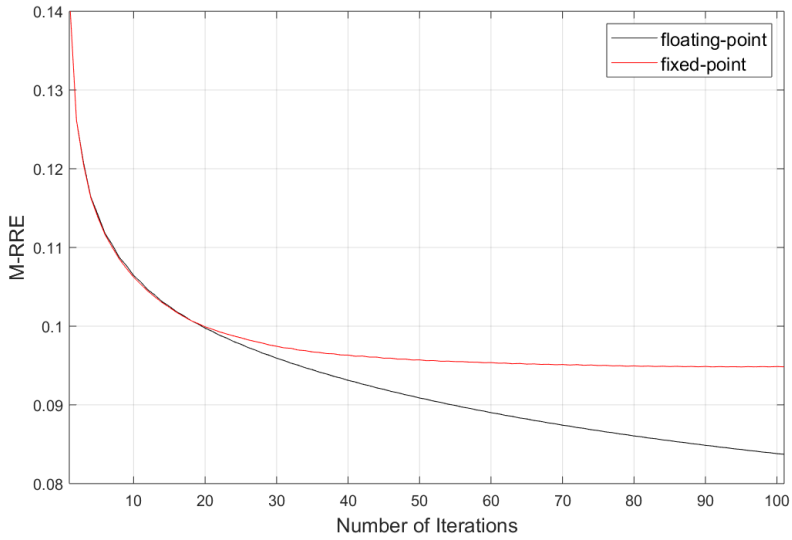


Figure 3.11: Comparison of a floating-point restoration error to the fixed-point restoration error.

In order to find a better solution, a MATLAB Fixed-Point Designer™ [48] is used. The floating-point data is stored in the fixed-point data objects fi

$$x_{fixed} = fi(x_{float}, sign, WL, FL) \tag{3.7}$$

where $sign$ can be either 0 for unsigned number or 1 for signed ones, WL is the word length and FL is the fractional length. The MATLAB function converting the floating-point RL-deconvolution to the fixed-point RL-deconvolution is written and MATLAB's Fixed-Point Designer™ proposed WLs and FLs are used in the updated C-script for fixed-point RL-deconvolution. The M-RRE plots using floating-point data, the blue plot, and fixed-point data, red plot, is shown in Figure 3.12. The achieved minimum squared error between the floating-point and fixed-point data is equal to MSE = 1.51 and the M-SSIM = 0.9997.

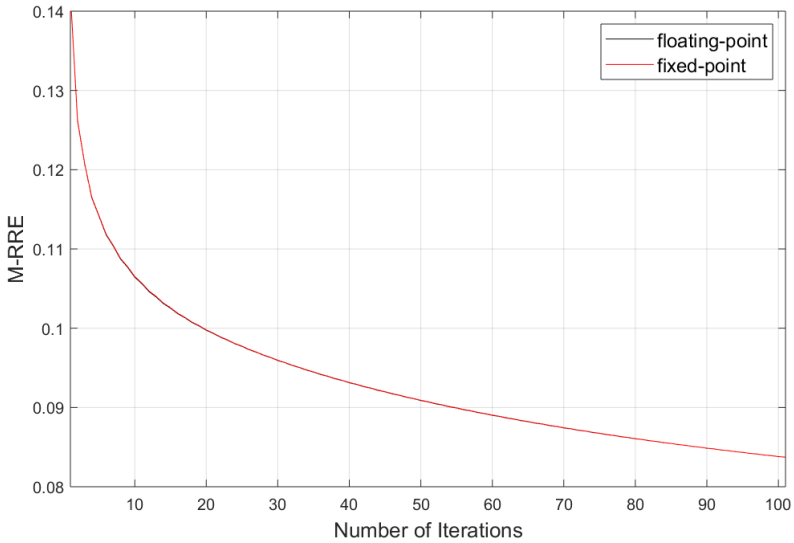


Figure 3.12: Comparison of a floating-point restoration error to the fixed-point restoration error.

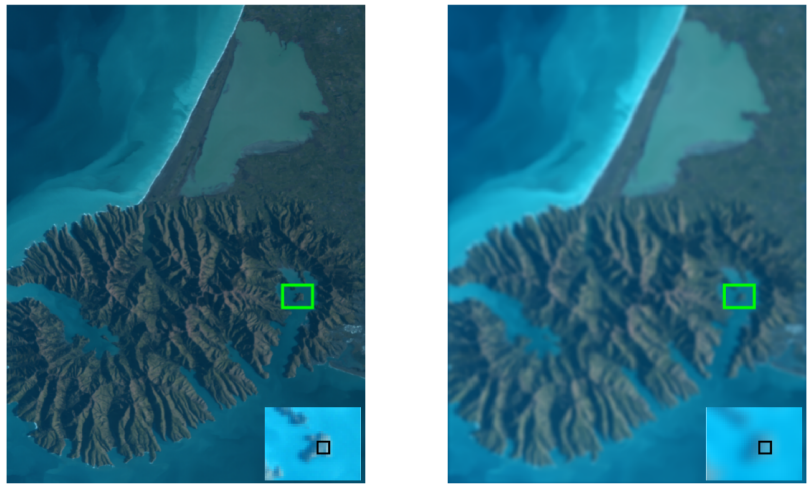
3.4 Hyperspectral Data - Coastal Image

The performance of the RL-deconvolution algorithm in a presence of noise is tested on the hyperspectral data from the Coastal dataset. The original hyperspectral cube is cropped in both the spectral and spatial directions, to the size equal to $640 \times 480 \times 60$. The spectral reduction is done in order to remove some of the corrupted bands and also to reduce the computation time. A three-band composition (18, 25, 53) of the original data, shown in Figure 3.13(a), is set as a reference, which is degraded with a degradation kernel equal to the Gaussian kernel with zero mean and $\sigma = 2.9$. The degraded image is shown in Figure 3.13(b). In addition, the degraded image is contaminated with an additive noise with SNR = 30 dB, shown in Figure 3.13(c) and SNR = 20 dB, shown in Figure 3.13(d). The M-PSNR and M-SSIM between the three degraded datacubes and the reference image are shown in Table 3.6, where Blurred stands for image degraded only by the Gaussian blur, Noisy1 denotes the image degraded by the Gaussian blur and an additive noise with SNR = 30 dB and Noisy2 refers to the image degraded by the Gaussian blur and an additive noise with SNR = 20 dB. The mean spectral reflectance plots are retrieved from the reference- and the degraded data cubes at a center sample with coordinates (396, 391). The total amount of samples used for the calculation of the mean spectral reflectance is equal to 25 samples. The area from which the reflectance plot is retrieved is marked by a black rectangular in Figure 3.13. The spectral reflectance plots are shown in Figure 3.14, which are used to analyze how the spectral reflectance is affected by the degradation and the restoration processes.

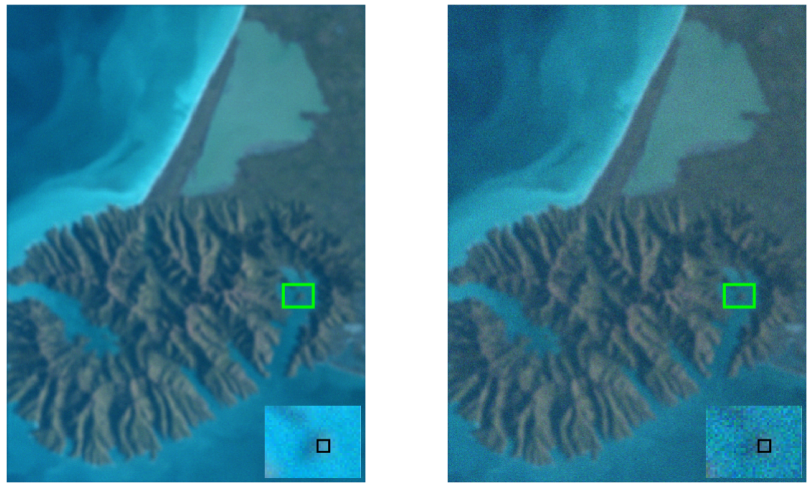
Table 3.6: M-PSNR and M-SSIM, for Blurred, Noisy1 and Noisy2 hyperspectral images compared with the reference image.

	M-PSNR	M-SSIM
Blurred	48.35	0.9654
Noisy1	48.19	0.9579
Noisy2	46.76	0.8989

The degraded data are restored using both the standard and accelerated RL-deconvolution algorithms. The VC-BCs are used when estimating the missing border information. Blurred, Noisy1 and Noisy2 datacubes are restored by running the RL-deconvolution $k = 50$ times and the restored datacubes are called Deblurred, Denoisy1 and Denoisy2, respectively. The M-RRE as a function of the number of RL-deconvolution iterations for all three data sets is shown in Figure 3.15. M-PSNR and M-SSIM are calculated at the minimum M-RRE and the results are shown in Table 3.7. The restored spectral reflectance plots are shown in Figure 3.16. Visual restoration results for the Deblurred, Denoisy1 and Denoisy2 images shown in Figure 3.17, Figure 3.18 and Figure 3.19, respectively.



(a) **(b)**



(c) **(d)**

Figure 3.13: Composition of three spectral bands (18, 25, 53) taken from the Costal hyperspectral dataset. The original three-band composite (a), used as a reference, is degraded with a Gaussian kernel with $\sigma = 2.9$ (b), which is further contaminated by an additive noise with SNR = 30 dB (c), or SNr = 20 dB (d).

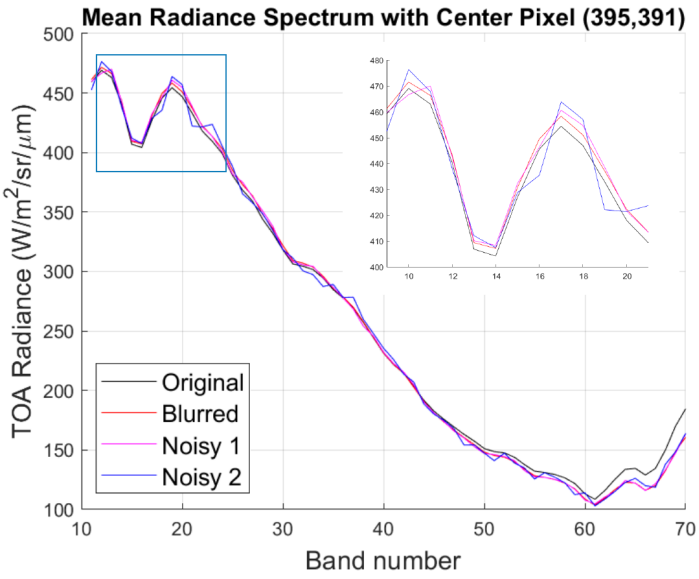


Figure 3.14: Mean Radiance Spectrum at the center sample (395, 391) retrieved from an area marked by the black rectangular in Figure 3.17.

Table 3.7: M-PSNR and M-SSIM at the minimum M-RRE reached after k standard RL-deconvolution iterations.

		M-PSNR(dB)		M-SSIM	
	k	M-PSNR	Relative improvement (dB)	M-SSIM	Relative improvement (%)
Deblurred	50	52.64	4.29	0.9809	1.61
Denoisy 1	27	50.99	2.80	0.9673	0.98
Denoisy 2	5	47.41	0.65	0.9009	0.22

Table 3.8: M-PSNR and M-SSIM at the minimum M-RRE reached after k accelerated RL-deconvolution iterations.

		M-PSNR(dB)		M-SSIM	
	k	M-PSNR	Relative improvement (dB)	M-SSIM	Relative improvement (%)
Deblurred	50	53.21	4.86	0.9835	1.87
Denoisy 1	14	50.98	2.79	0.9673	0.98
Denoisy 2	2	47.43	0.67	0.9014	0.29

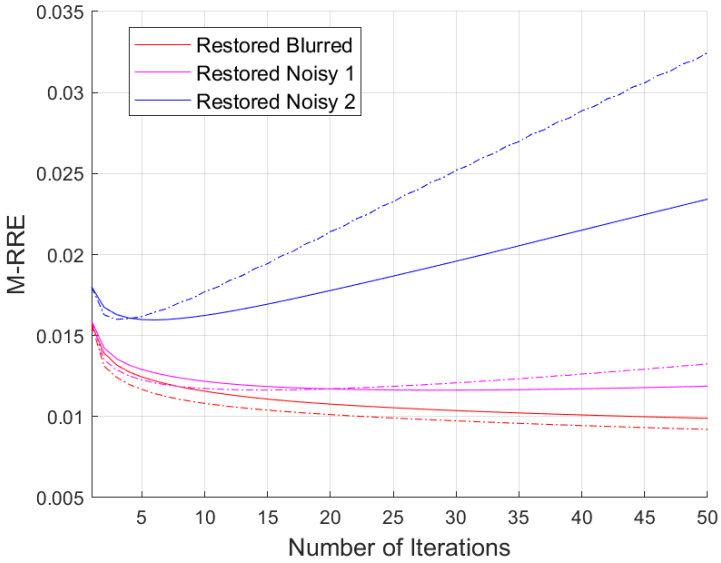


Figure 3.15: M-RRE as a function of the number of iterations for dataset degraded with an additive noise.

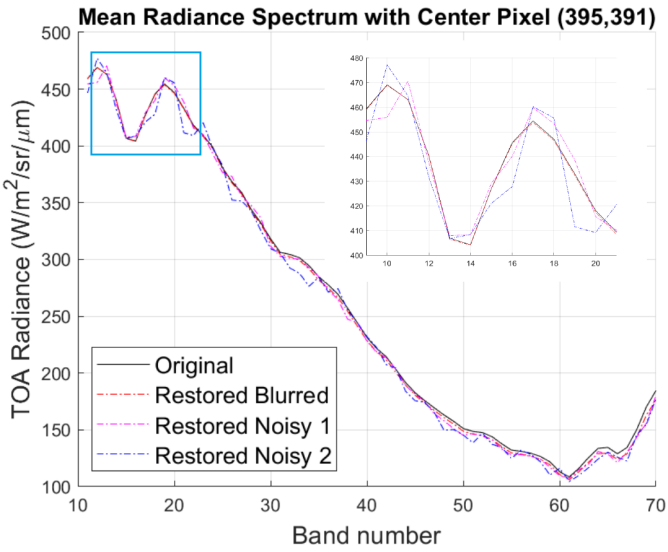


Figure 3.16: Mean Radiance Spectrum at center sample (395, 391) retrieved from an area marked by the black rectangular in Figure 3.17.

(a) Reference image.

(b) Degraded with $\mathbf{H}_D(\sigma) = 2.9$. M-PSNR=48.35, M-SSIM=0.9654.

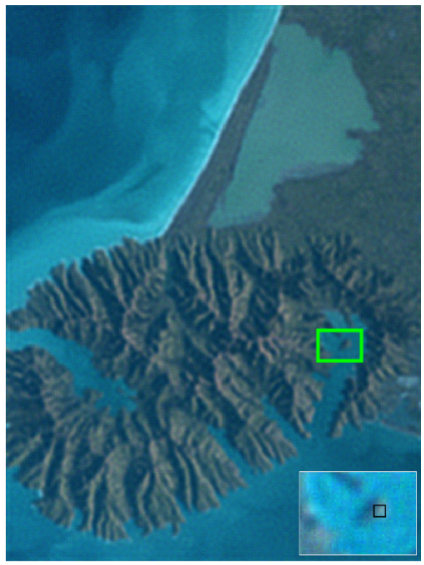
(c) $\mathbf{H}_R(\sigma) = 2.9$, $k = 50$. M-PSNR=53.21, M-SSIM=0.9835.

(d) $\mathbf{H}_R(\sigma) = 2.9$, $k = 500$. M-PSNR=55.15, M-SSIM=0.9913.

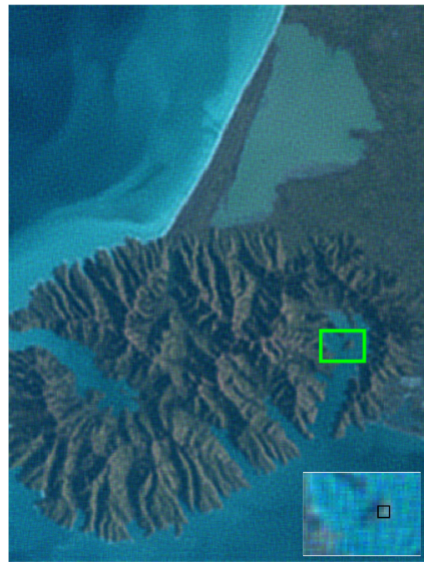
Figure 3.17: The degraded by the degradation kernel $\mathbf{H}_D(\sigma) = 2.9$ image (b) is restored with $\mathbf{H}_R(\sigma) = 2.9$ for $k = 50$ RL-deconvolution iterations (c) and $k = 500$ RL-deconvolution iterations (d). The close-up of the area marked by the green rectangular is shown in the left corners of the images.

(a) Reference image.

(b) $\mathbf{H}_D(\sigma) = 2.9$ + SNR 30 dB.
M-PSNR=48.19, M-SSIM=0.9579.



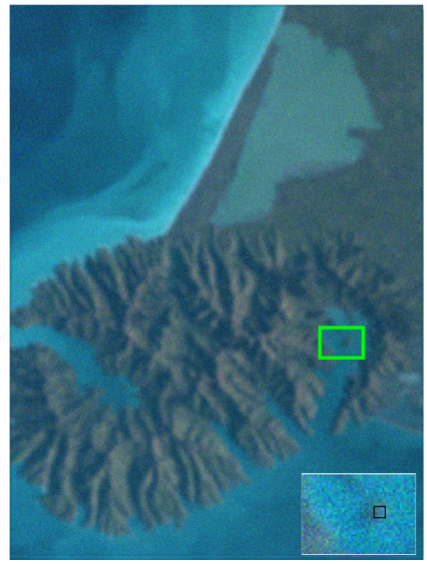
(c) $\mathbf{H}_R(\sigma) = 2.9$, $k = 14$.
M-PSNR=50.98, M-SSIM=0.9579.



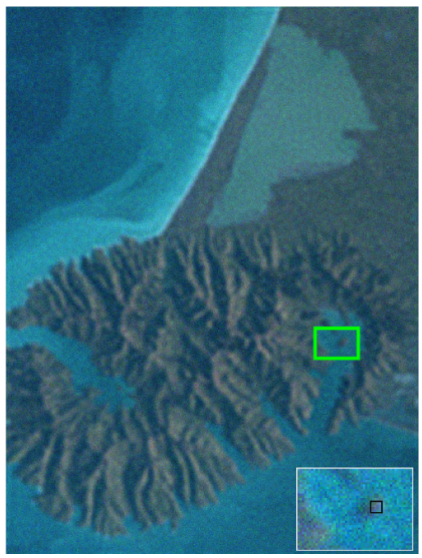
(d) $\mathbf{H}_R(\sigma) = 2.9$, $k = 50$.
M-PSNR=49.56, M-SSIM=0.9512.

Figure 3.18: The degraded by the degradation kernel $\mathbf{H}_D(\sigma) = 2.9$ and the Gaussian noise with SNR = 30 dB image (b) is restored with $\mathbf{H}_R(\sigma) = 2.9$ for $k = 14$ RL-deconvolution iterations (c) and $k = 50$ RL-deconvolution iterations (d).The close-up of the area marked by the green rectangular is shown in the left corners of the images.

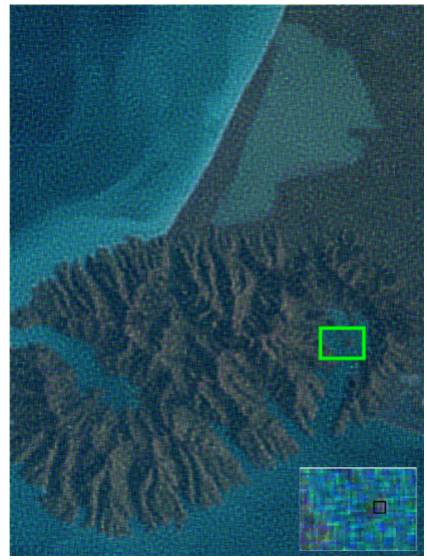
(a) Reference image.



(b) $\mathbf{H}_D(\sigma) = 2.9$ + SNR 20 dB.
M-PSNR=46.78, M-SSIM=0.8989.



(c) $\mathbf{H}_R(\sigma) = 2.9$, $k = 2$.
M-PSNR=47.43, M-SSIM=0.9014.



(d) $\mathbf{H}_R(\sigma) = 2.9$, $k = 50$.
M-PSNR=41.82, M-SSIM=0.7630.

Figure 3.19: The degraded by the degradation kernel $\mathbf{H}_D(\sigma) = 2.9$ and the Gaussian noise with SNR = 20 dB image (b) is restored with $\mathbf{H}_R(\sigma) = 2.9$ for $k = 2$ RL-deconvolution iterations (c) and $k = 50$ RL-deconvolution iterations (d). The close-up of the area marked by the green rectangular is shown in the left corners of the images.

As seen in Figure 3.17 and in Table 3.7 and Table 3.8, the presence of additive noise prevents the restoration of the degraded images. When SNR = 30 dB, the M-PSNR and M-SSIM has a relative improvement equal to 2.80 dB and 0.98 %, respectively, compared to the Noisy1 data. When SNR = 20 dB, the M-PSNR and M-SSIM has a relative improvement equal to 0.65 dB and 0.22 %, respectively, compared to the Noisy2 data. An improvement, although not signifact, is present in both cases, as long as the algorithm is stopped when the minimum M-RRE is reached. As seen in Figure 3.15, the M-RRE starts diverging after around 14 to 27 iterations for Noisy1 data and after around 2 to 5 iterations for Noisy2 data. This is also visible in Figure 3.18(d) and Figure 3.19(d), showing the deconvolution result after 50 iterations for Denoisy1 and Denoisy2 respectively. In this case, the M-PSNR and M-SSIM has a relative improvement equal to 1.37 dB and -0.70 %, respectively, compared to the Noisy1 data and M-PSNR and M-SSIM equal to -4.96 dB and -15.12 %, respectively, compared to the Noisy2 data.

The spectral reflectance plots for the degraded data, shown in Figure 3.14, is compared to the spectral reflectance plots for the restored iterations data, shown in Figure 3.16. The spectral reflectance of the data degraded only by the Gaussian blur is fully restored at the sample with coordinates (395, 391). The spectral reflectance of the data degraded by the Gaussian blur and an additive noise, remains unchanged after the restoration. It should be noted that in all cases the spectral reflectance plot keep its original shape.

Chapter 4

Hardware Implementation

4.1 Introduction

One RL-deconvolution iteration consists of two convolutions, one element-wise division and either one element-wise multiplication, in the case of the standard RL-deconvolution algorithm (equation (2.7)), or two element-wise multiplications, in the case of the accelerated RL-deconvolution algorithm with $\beta = 2$ (equation (2.9)). Both versions can be implemented into one design without a significant increase in the resource usage, as only one additional multiplier is needed. The degraded input image, $\mathbf{Y}^p$, is used in each iteration in order to calculate the residual and it needs to be either stored in the internal memory of the FPGA, or streamed from the external memory for each new iteration. The former minimizes the communication time between the FPGA and the external memory, while the latter minimizes the internal memory usage. Two architectures exploring both methods are designed, henceforth called *Architecture-1* and *Architecture-2*. The two architectures are designed to perform both the standard RL-deconvolution and the accelerated RL-deconvolution with $\beta = 2$, where the end-user has the possibility to chose at the run-time which of the two algorithms to use. A summarized overview of main features for Architecture-1 and Architecture-2 is shown in Table 4.1.

Table 4.1: Feature summary for Architecture-1 and Architecture-2.

Feature	**Architecture-1**	**Architecture-2**
Standard RL-deconvolution	✓	✓
Accelerated RL-deconvolution	✓	✓
Initial value, $\hat{\mathbf{X}}^p_{(0)}$	$\mathbf{Y}^p$	constant
VC-BCs	✓	
MP-BCs		✓
Generic image size	✓	✓
Run-time conf. kernel size		✓
Run-time conf. number of iterations	✓	✓
Independent of DDR3	✓	

4.2 Architecture-1

A generalized block diagram for Architecture-1 is shown in Figure 4.1. It consists of three blocks, the *Core IP*, for a complete RL-deconvolution algorithm, an external *Division IP* and a *Configuration* block with an AXI4-Lite Slave interface. Table 4.2 lists the signals, directions and corresponding short descriptions for the inputs and outputs for the RL-deconvolution IP. Only the signal, `s00_axi_wdata`, is shown for the AXI4-Lite channels. Table 4.3 shows the list of generic parameters.

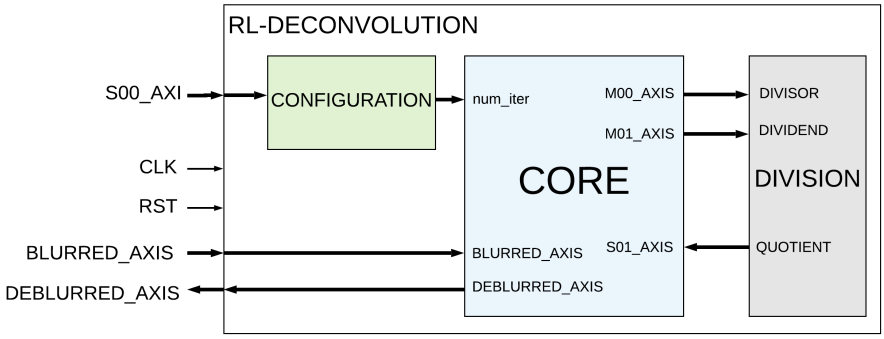


Figure 4.1: A block diagram for the top module of RL-deconvolution IP.

Table 4.2: List of Input/Output signals for the RL-deconvolution IP.

Name	I/O	Width	Description
`s00_axi_wdata`	in	32	wadata signal for s00_axi channel.
`blurred_axis_tdata`	in	16/32	tdata signal for blurred_axis channel.
`blurred_axis_tlast`	in	1	tlast signal for blurred_axis channel.
`blurred_axis_tvalid`	in	1	tvalid signal for blurred_axis channel.
`blurred_axis_ready`	out	1	tready signal for blurred_axis channel.
`deblurred_axis_tdata`	out	16/32	tdata s. for deblurred_axis channel
`deblurred_axis_tlast`	out	1	tlast s. for deblurred_axis channel
`deblurred_axis_tvalid`	out	1	tvalid s. for deblurred_axis channel
`deblurred_axis_ready`	in	1	tready s. for deblurred_axis channel
`clk`	in	1	Global clock high
`rst`	in	1	Global active reset.

Table 4.3: Generic parameter list for the RL-Deconvolution IP.

Name	Sym.	Description
`IMAGE_HEIGHT`	N	The height of the input data.
`IMAGE_WIDTH`	M	The width of the input data.
`DATA_WIDTH`	W_D	Input/output data width.
`ADDR_WIDTH`	W_A	Address width, depends on image size.

4.2.1 Core IP

The Core IP, shown in Figure 4.1, consists of a data path and a control path. The block diagram for the data path together with the connected Division IP is shown in Figure 4.2. The design is scalable with respect to the image size and run-time configurable with respect to the number of iterations for RL-deconvolution algorithm. The goal of the design is to minimize the communication between the external memory and the programmable logic, by storing all the needed data inside the FPGA accelerator. In order to internally store large amount of data, Xilinx *Block* RAMs (BRAMs) are used, where one block can store up to 36 Kb of information [49]. Before doing the convolution, the input image is extended in order to preserve its initial size. The image borders are estimated using the VC-BCs, described in Section 2.1.3, as it gives the best visual results. Convolution is then done by moving a whole kernel from one row to another, as shown in Figure 2.2. The value of the estimated samples is stored in the REG_0 and is equal to the first value of either the data stored in $BRAM\_F$ or $BRAM\_T$, depending on which data is sent to the convolution module.

One iteration of the RL-deconvolution algorithm proceeds as follows: the degraded input image, $\mathbf{Y}^p$, is streamed from the external memory into the Core IP through the AXI-Stream `blurred_axis` input and is stored in the $BRAM\_G$ and $BRAM\_F$. The address generators ensure the correct reading and writing to/from the BRAMs. The data stored in the $BRAM\_F$ sets the initial estimate, $\hat{\mathbf{X}}^p_{(0)}$, to be equal to the blurred input image, $\mathbf{Y}^p$. After the input streaming is finished and signal `blurred_axis_tlast` is received the module is left to be independent of the rest of the system, meaning that no control from the PS side is needed. Immediately after the signal `blurred_axis_tlast` is received, the convolution module is enabled, where convolution is done on either the data

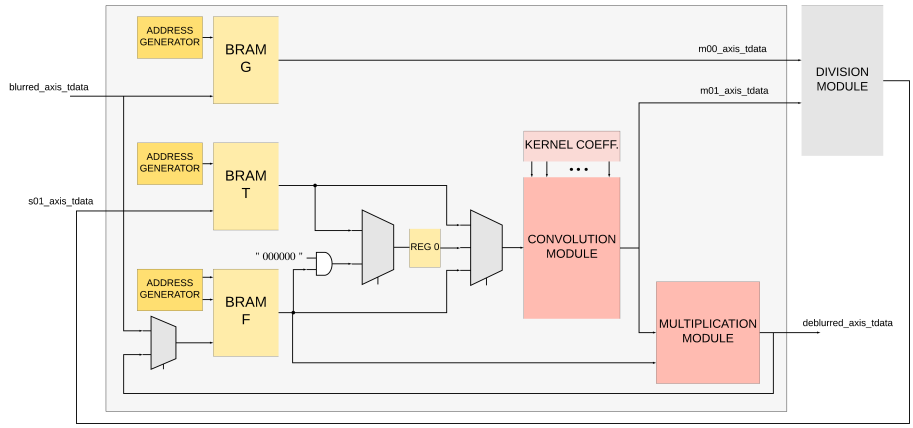


Figure 4.2: Block diagram for Architecture-1.

stored in REG_0, or on the data stored in $BRAM\_F$ (depending if the image itself or its borders are being processed). Division starts immediately after the first valid output sample from the convolution module is produced. The dividend is equal to the data stored in $BRAM\_G$ corresponding to the distorted input image and the divisor is equal to the convolution output data. The division output is stored in $BRAM\_T$. The second part of the algorithm starts immediately after the signal `s01_axis_tlast` from the division core is received. Convolution is then done on the data stored in either REG_0 or in the $BRAM\_T$. The valid output samples from the convolution module are multiplied with the old estimated value $\hat{\mathbf{X}}^p_{(k-1)}$ (i.e., $\hat{\mathbf{X}}^p_{(0)}$) stored in $BRAM\_F$. Finally, the data in $BRAM\_F$ is updated with the new estimated value, $\hat{\mathbf{X}}^p_{(k)}$. If several iterations are intended, the process repeats, else the multiplication output is streamed out through the AXI-Stream `deblurred_axis` channel.

4.2.2 Data Precision

The core is developed for input data width equal to 16 bits. In order to preserve the output data precision, the fractional part of the quotient is set to be equal to 20 bits, as it was found in Subsection 3.3.3. The quotient integer part is always between 0 and 1, thus it is sufficient to use one bit for the integer part. It is chosen to use 22 bit width for the division output data. Thus $BRAM\_T$ stores the data with width 22 bits, while $BRAM\_G$ and $BRAM\_F$ stores data with width 16 bits. The convolution is done on the data with width 22 bits, therefore zeros are appended to the data from the $BRAM\_F$. The multiplication is done on the 22 bit width data from the convolution and 16 bit wide data from the $BRAM\_F$.

4.2.3 Convolution Module

Convolution on a 2-D image of size $N \times M$ with a kernel of size $W_x \times W_y$ is performed by a convolution module proposed by [50], where $W_x = W_y = W$. The 2-D convolution is accelerated by exploiting kernel separability, where a 2-D kernel is decomposed into two 1-D kernels. This allows a design with two 1-D convolutions, where a convolution with a 1-D vertical kernel, $\mathbf{H}_y$, on the continuous stream of input data is followed by a convolution with a 1-D horizontal kernel, $\mathbf{H}_x$.

Convolution is done by sliding the input image through the kernel, where at each position a sample spanned by the kernel is processed by equation (2.5). An image in Figure 4.3 illustrates computation of the first two output samples for a 1-D convolution of an input image of size 5×5 with a kernel of size 3×1. The hardware module receives the matrix in Figure 4.3 in a continuous data stream. In order to produce, the first output sample, the samples from the first two rows (i.e., elements $X_{(0,0)}$ to $X_{(2,0)}$) needs to be streamed into the convolution module. For the second output element (i.e., elements $X_{(0,1)}$ to $X_{(2,1)}$) are needed. It can be observed that some samples are being reused. In order to avoid the reloading of the same sample into the convolution module, line buffers are used.

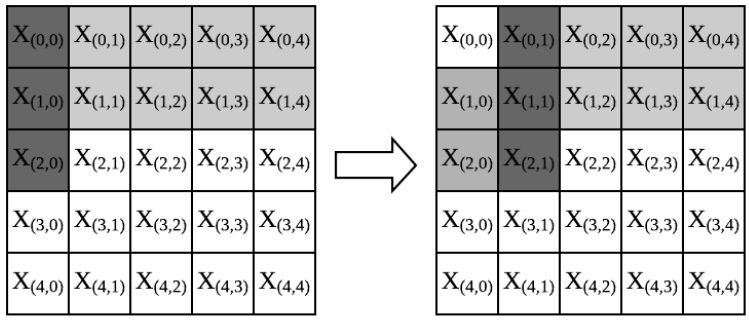


Figure 4.3: An example of data requirements for 1-D convolution with a 3×1 kernel.

The block diagram, consisting of line buffers and two 1-D convolutions for a kernel of size 3, is shown in Figure 4.4. The first input sample $X_{(0,0)}$ enters the convolution module and is stored in a register (called $DATA\_1$ in Figure 4.4). On the next clock cycle $X_{(0,0)}$ enters the first convolution with a vertical kernel, and at the same time is sent to the first line buffer (called $LINE\_1$ in Figure 4.4) and the second input element, $X_{(1,0)}$ gets stored in the $DATA\_1$, shown in Figure 4.5(b). The size of the line buffer depends on the width of the input image and is equal to $N - 1$. For this example, the width of the line buffer is equal to 4. When the first line buffer is filled, as illustrated in Figure 4.5(c), the data from

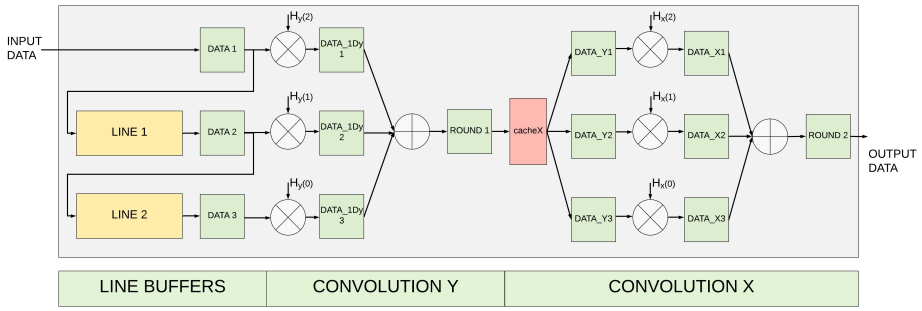
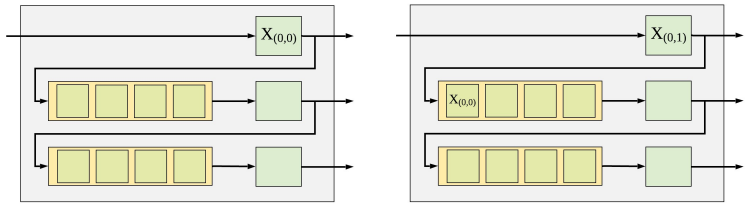


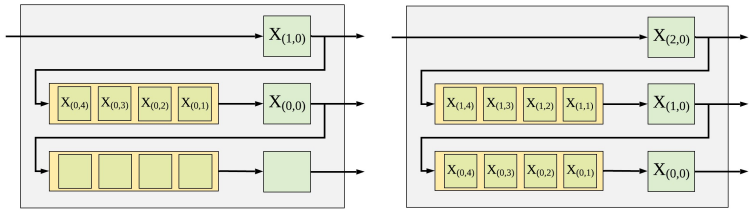
Figure 4.4: An block diagram for the 2-D convolution with a separable 3×3 kernel.

the $LINE\_1$ starts moving to the $LINE\_2$. The total number of line buffers depends on the kernel size and is equal to $W - 1$, which for this example is equal to 2. At the end of the initialization, the first row of the input image will be stored in the $LINE\_2$ and the second row of the input image will be stored in the $LINE\_1$, as shown in Figure 4.5(d).



(a) The first input pixel is streamed into the accelerator.

(b) The second input pixel enters the accelerator. The first input pixel moves to the first line buffer.



(c) Filled first line buffer.

(d) The end of the initialization stage.

Figure 4.5: Initialization of the line buffers.

After initialization, the line buffer outputs has the elements corresponding to the dark grey data elements in Figure 4.3, as seen in Figure 4.5(d). These elements are multiplied in parallel with their corresponding kernel values, summed and sent to the CacheX block shown in Figure 4.4. The CacheX block concatenates the resulting data from first convolution into one array of the same size as the

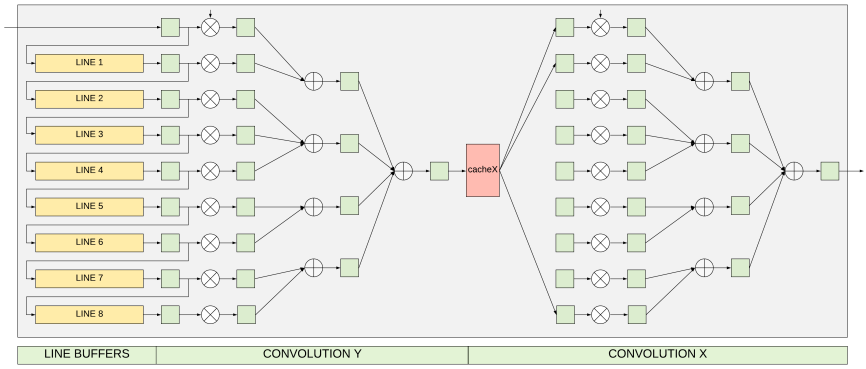


Figure 4.6: A block diagram for the 2-D convolution with a separable 9×9 kernel.

kernel. This is done in order to be able to do a convolution with a horizontal kernel in parallel. For example, for the left image in Figure 4.3, the concatenated array will have elements $[X_{(0,0)}, X_{(0,1)}, X_{(0,2)}]$. The CacheX block sends the array elements to the corresponding registers $DATA\_Y\#$ shown in Figure 4.4. The element $X_{(0,0)}$ goes into register $DATA\_Y3$, the element $X_{(0,1)}$ goes into register $DATA\_Y2$ and element $X_{(0,2)}$ goes into register $DATA\_Y1$. These values are then multiplied and summed. The resulting output data is rounded.

The convolution module implemented in this design was made for a kernel size equal to 9×9 and it follows the same principle as described above. The kernel coefficients are read from outside of the convolution module. The addition is done by a pipelined adder tree. The block diagram is shown in Figure 4.6.

4.2.4 Multiplication Module

The block diagram for the multiplication module is shown in Figure 4.7. The `INPUT_1` is connected to the convolution module output with the signal width equal to 22 bits and `INPUT_2` is connected to the $BRAM\_F$ with the signal width equal to 16 bits. This module is responsible for choosing between the standard- and the accelerated RL-deconvolution. This is done with the help of the MUX_1, which is set by the end-user through the Configuration block. For the standard RL-deconvolution, the `INPUT_1` is multiplied with `INPUT_2`, delayed for three clock cycles and shifted to the right by 14 bits. For the accelerated RL-deconvolution, the result of the first multiplication is multiplied again with the delayed `INPUT_1`, as explained in Chapter 2. The result is shifted a different amount of bits, depending on which iteration is run. For the first iteration the result is shifted 16 bits to the right, while for the rest of the iterations the result is shifted 20 bits to the right. The values are found experimentally, during

the analysis part of the implementation described in Chapter 3. MUX_0 is controlled by the internal controller.

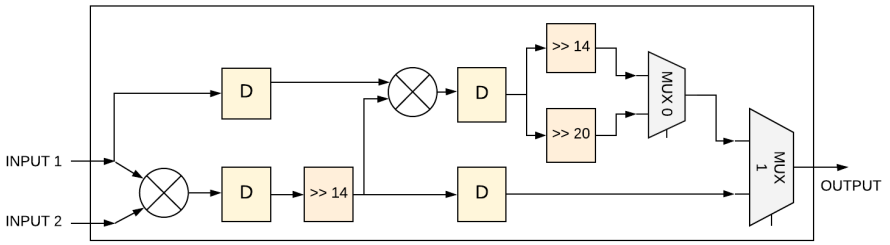


Figure 4.7: Multiplication module. D stands for the delay.

4.2.5 Division Core

Division is done using a Pipelined Divider v5.1 core provided by Xilinx [5]. The core is customized through the Core Generator tool. There are three possible division implementations, i.e., **LUTMult**, **Radix-2** and **High Radix**, each having their advantages and disadvantages. The detailed description of each is given in [5]. The **Radix-2** implementation is chosen as it does not use the DSP nor BRAM resources and also provides a high throughput. For a fully pipelined design, a throughput of one division per clock cycle is achieved. When a fractional output is required, the latency is equal to $M + F$, where M is the quotient width and F is the fractional output width. The block diagram with the input/output signals is shown is Figure 4.8.

An M-bit dividend is divided by an N-bit divisor producing a quotient with a fractional remainder. For the unsigned case, the fractional part is equal to

$$FractRmd = \frac{IntRmd \times 2^F}{Divisor}. \tag{4.1}$$

The quotient bit width is equal to the divisor width and the fractional output width is set independently. The core uses a global synchronous, active-low reset which must be asserted for at least two clock cycles. The divider core uses AXI-Stream interface to communicate with other modules. The only non-optional signals are `s_axis_dividend_tvalid`, `s_axis_dividend_tdata`, `s_axis_divisor_tvalid`, `s_axis_divisor_tdata`, `m_axis_dout_tvalid` and `m_axis_dout_tdata`. All other signals seen in Figure 4.8 are optional. The tlast signal can be used in order to avoid matching latency to the data path. The tlast is read from the slave modules and is sent to the output channel with the same latency as the datapath.

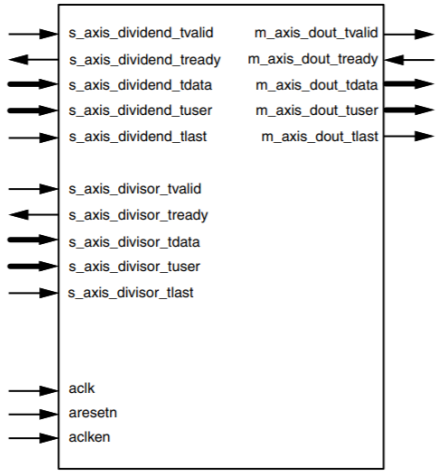


Figure 4.8: The pinout diagram for Pipelined Divider v5.1 core [5].

4.2.6 Configuration Module

The Configuration block seen in Figure 4.1 contains a configuration register which is used to configure the performed algorithm. The register is 32 bits wide and written using the AXI4-Lite interface. The MSB is reserved for the algorithm-select signal, where 1 selects the standard RL-deconvolution and 0 selects the accelerated RL-deconvolution. The remaining 31 bits assign the number of iterations for RL-deconvolution.

4.2.7 Control Path

The RL-deconvolution architecture contains one convolution module, which is used twice during one iteration of the algorithm. Convolution output is connected to either the division IP or the multiplication IP. In addition, the convolution is performed as shown in Figure 2.2, i.e., the whole kernel is moved from one row of the image to another. This adds some additional delay for the output data stream from the convolution module, which in turn adds a delay in the output from both the multiplication and division modules. Therefore, in order to simplify the control system, the control of the architecture is split into two stages: convolution and division (done by the Convolution&Division controller), and convolution and multiplication (done by the Convolution&Multiplication controller). A simplified data block diagram is shown in Figure 4.9, where the master controller connects the two stages together. The state diagram for the master controller is shown in Figure 4.10 and the corresponding inputs/output signal name are shown in Table 4.4. The numbering of the bit arrays seen in Figure 4.10 corresponds to the numbering in Table 4.4, where 0 is the LSB. The controller is a Moore finite state machine, where outputs depend solely on the state, therefore in Figure 4.10, outputs are shown inside the states and inputs are placed on transitions. All states are connected to the **IDLE** state, which is entered if the controller is reset. The state diagrams for Convolution&Division and Convolution&Multiplication controllers can be found in Appendix A. The control module, in addition to internal control signals connected to the data path, is also connected to the external AXI-Stream control signals.

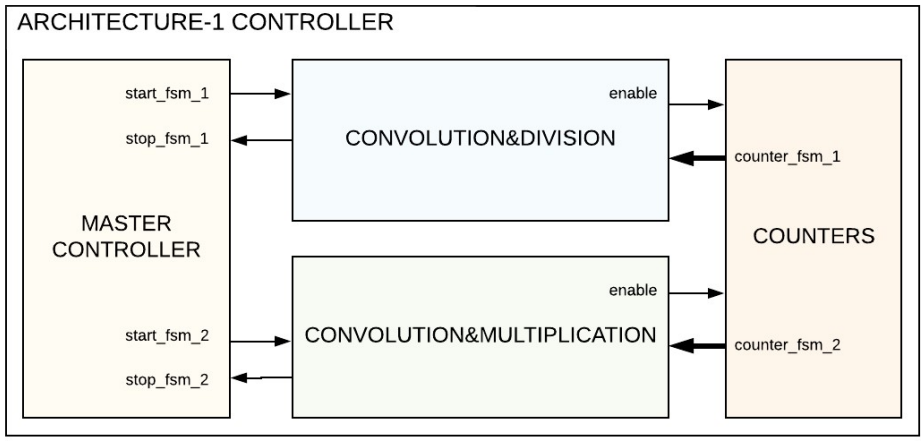


Figure 4.9: A simplified block diagram of the control path for Architecture-1.

The RL-deconvolution module starts when the `blurred_axis_tvalid` is equal to 1. The master controller enters the state **INIT**, in which the $BRAM\_F$ and $BRAM\_G$ are initialized with the input data. When the last input image element is received and the `blurred_axis_tlast` is equal to 1, the REG_0,

shown in Figure 4.2, is initialized with the first value stored in $BRAM\_F$, state **PAD1**, and after two clock cycles the first stage, Convolution&Division, is entered, state **CONV&DIV**, where Convolution&Division controller is enabled. Convolution&Division controller is responsible for correctly choosing the data to be sent to the convolution module (i.e., either data from REG_0 or data from $BRAM\_F$) and also sending the valid output samples to the division module. The master controller stays in this state until the signal `stop_fsm_1` is received from the Convolution&Division controller. The master controller moves to the state **PAD2**, where the REG_0 is initialized with the first value in $BRAM\_T$. The second stage, Convolution&Multiplication is entered and a signal `start_fsm_2` is set to 1, starting the Convolution&Multiplication controller. In this case, Convolution&Multiplication controller sends the data either from REG_0 or data from $BRAM\_T$ to the convolution module and also sends the valid output samples to the multiplication module. When the `stop_fsm_2` is received, the state **FIN2** is entered, where if the signal *iter* is equal to 1, the master controller goes back to the **IDLE** state, else if *iter* is equal to 0, then the master controller goes back to the state **PAD1**, where it re-initializes the $REG0$ and the process repeats.

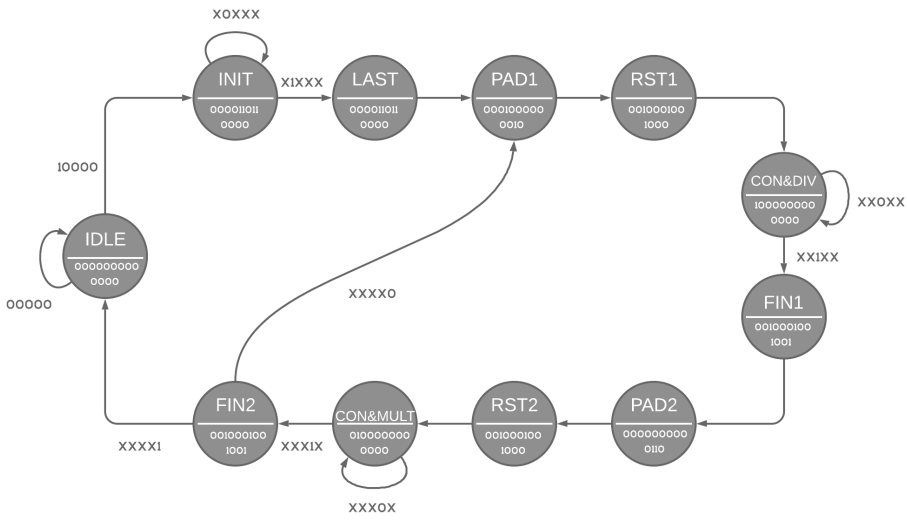


Figure 4.10: State diagram for Architecture-1's master controller.

Table 4.4: Input/output signal names corresponding to the state diagram of the master controller.

Bit	Input	Bit	Output
0	blurred_axis_tvalid	0	start_fsm_1
1	blurred_axis_tlast	1	start_fsm_2
2	stop_fsm_1	2	reset_bram0
3	stop_fsm_2	3	enb_0
4	iter	4	ena_0
		5	we_0
		6	reset_bram1
		7	ena_1
		8	we_1
		9	reset_bram2
		10	ena_2
		11	enable_pad_reg
		12	reset_modules

4.3 Architecture-2

A generalized block diagram for Architecture-2 is shown in Figure 4.11. The top module is similar to the Architecture-1, containing three blocks, the *Core IP*, containing both the datapath and the control system, an external *Division IP* and an *Configuration* block with an AXI4-Lite Slave interface. The inputs and outputs for the RL-deconvolution IP and generic parameters are the same as for the Architecture-1, and are shown in Table 4.2 and Table 4.3 respectively. The only difference between Architecture-1 and Architecture-2 on the top level is two additional signals, `enable_module` and `sigma_and_iter`.

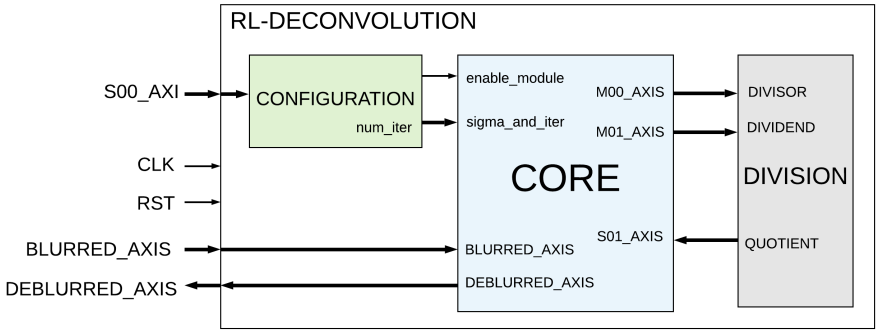


Figure 4.11: A block diagram for Architecture-2.

4.3.1 Core IP

This architecture is designed to deal with limited internal FPGA storage. The solution uses one True Dual-Port BRAM to store the latent image, $\hat{\mathbf{X}}^{p}_{(k)}$. An additional convolution module is added in order to avoid storing the intermediate data produced by the division module. The degraded input image, which remains constant during the deconvolution process, is streamed from the external memory instead of being stored inside the module. The block diagram for the complete RL-deconvolution is shown in Figure 4.12.

The module is parametarized with respect to the input image size. In addition, the module is run-time configurable with respect to the kernel size and the number of RL-deconvolution iterations. The core is designed to work with a kernel size equal to 9×9 and 7×7. There is a Read-Only memory initialized with a few different kernels having a varying sigma. The end user can choose the σ by simply writing the wanted value, together with the number of iterations, into the register stored in the Configuration block, shown in Figure 4.1. The data processing starts immediately after the signal `enable_module`, which is received

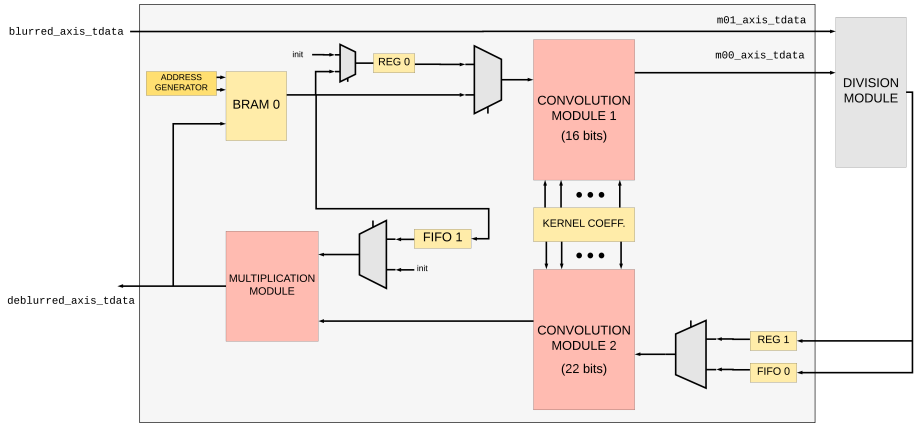


Figure 4.12: Complete block diagram for RL-deconvolution.

from the Configuration block. The first iteration of the algorithm is done on the constant predefined value, which is set to be equal to 1000. This value was chosen experimentally. For this architecture, MP-BCs are chosen, giving reasonably good visual results, as seen in Section 2.1.3. In this case convolution is done as show in Figure 2.3, where for a continuous stream of input data, the kernel always is moved by constant shift. This simplifies the controller system.

4.3.2 Configuration Module

The architecture of the configuration block is the same as for the Architecture-1. The difference is the use of the signal `s00_axi_wdata`. In this case, the 32 bit signal is divided as follows: the first 16 bits before the LSB are reserved for the total number of iterations, the MSB is reserved for the algorithm-select signal and the 15 bits after the MSB are used for writing restoration kernel size.

4.3.3 Multiplication module

Multiplication core is the same as the one used for Architecture-1.

4.3.4 FIFO modules

In Figure 4.12, an additional *First In, First Out* (FIFO) block is added. FIFO is a data buffer, where the first entry, the oldest one, is processed first. In this design, FIFO is implemented using Xilinx FIFO Generator v13.1 [51] with an

AXI-Stream interface. For the simplicity of the illustration, FIFOs in Figure 4.12 are shown inside the Core IP, when in reality they are instantiated outside the core and are connected to the core in a same way as the external Division IP.

There are two FIFOs in the design, $FIFO\_0$ and $FIFO\_1$. Convolution module 2 in Figure 4.12 processes the data stored in REG_1 first, and then the data from the division IP. The division output samples need to be stored until the processing of the the data in REG_1 is done. The depth of $FIFO\_0$ depends on the image width and the kernel size, and is equal to

$$\text{FIFO\_0}_{depth} = M \times ((W-1)/2) + 3. \tag{4.2}$$

where M is the image width and W is the kernel size. The true Dual-Port BRAM used in this design has one read and one write port which can be used simultaneously. In this design, two read ports and one write port with different addresses are needed. To solve this problem a $FIFO\_1$ is added. The data from $BRAM\_0$ enters $FIFO\_1$ at the same time as it enters the Convolution module 1. The depth of the $FIFO\_1$ is equal to

$$\text{FIFO\_1}_{depth} = M \times (3 \times (W-1) + (W_{(max)} - 1)) + 52 \tag{4.3}$$

where $W_{(max)}$ is the maximum available kernel size, in this architecture $W_{(max)}$ is equal to 9.

4.3.5 Convolution Module

The convolution module is a more flexible version of the one designed for the Architecture-1. The module has a fixed maximum kernel size, which is chosen to be equal to 9×9, in this case the data flow is the same as shown in Figure 4.6. By using a control signal, `sigma`, a kernel size smaller than 9×9 can be chosen. Assuming, the kernel size is equal to 7×7, then the data out from the register $DATA\_1$ and the data from the first line buffer $LINE\_1$ are going to be multiplied with zeros. Then, the result from the first convolution is concatenated into an array in CacheX, shown in Figure 4.6, the size of which depends on the kernel size. As mentioned previously, this is done in order to be able to perform convolution with a horizontal kernel in parallel. The data from the CacheX is sent to the corresponding registers as shown in Figure 4.6. The fixed number of which is equal to 9. In the case when the kernel size is equal to 7, the two extra registers will be initialized with zeros.

4.3.6 Control Path

There are four controllers taking care of the algorithm's data flow: two convolution controller, division controller and multiplication controller. An additional master controller ensures the correct enabling of each of them. A simplified block diagram showing the complete control path is shown in Figure 4.13. Control module is also connected to the global clock and reset signals. All states are connected to the **IDLE** state, which is entered if the controller is reset.

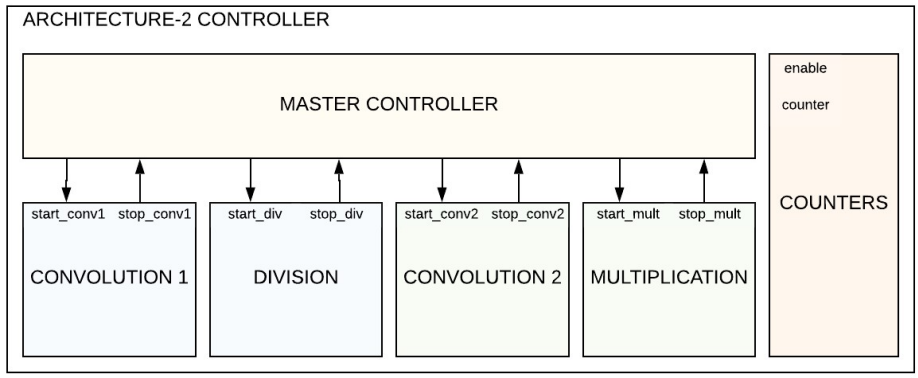


Figure 4.13: Block diagram for the control path of the Architecture-2.

Counter module, shown in Figure 4.13, contains four counter instances, each connected to one of the main controllers. The state diagram for the master controller is shown in Figure 4.14 with the corresponding inputs/output signal names in Table 4.5. The state diagrams for the Convolution 1 controller, Division controller, Convolution 2 controller and Multiplication controller are shown in Figure 4.15, Figure 4.16, Figure 4.17 and Figure 4.17 respectively. Their corresponding inputs/output signal names are shown in Table 4.6, Table 4.7, Table 4.8 and Table 4.9.

The control path designed for the Architecture-2 is somewhat simpler than the one designed for the Architecture-1. This is due to the use of different boundary conditions. In Architecture-1, VC-BCs are used, while in Architecture-2, MP-BCs are implemented. The latter simplifies the controller system, due to the way convolution of the whole image is done. In this case, the kernel is always shifted periodically, which results in all modules producing one valid output sample per clock cycle. The master controller is enable when the signal `enable_module` is received. In the state **CONV1**, the Convolution 1 controller is enabled and convolution on the data in either REG_0 or in $BRAM\_0$ is done. After the initial latency of the Convolution module, which depends on the image and kernel sizes, the state **DIV** is entered and the Division controller is enabled. Master controller stays in the state **DIV** until the signal `s01_axis_tvalid` is received, indicating that the division output data is valid and ready to be processed. The

second convolution module is enable in the state **CONV2**, where convolution is done on the division output data. When a first valid output sample from the convolution module is produced, the master controller enters the state **MULT**, at which point the last step of the algorithm is performed. Finally, in state **FIN**, the signal *iter* is checked, if it is equal to 1 then the master controller goes to the state **IDLE**, else the master controller enters the state **PAD1** and the described process repeats.

The constants seen in Figure 4.15, Figure 4.17 and Figure 4.18 are given by

$$\text{PAD\_1} = (W-1)/2 \times N \tag{4.4}$$
$$\text{START\_DIV} = \text{PAD\_1} + \ N \times M \tag{4.5}$$
$$\text{PAD\_2} = \text{START\_DIV} + \text{PAD\_1} \tag{4.6}$$
$$\text{FIN\_IMG} = (W-1) \times N + W + 4 \tag{4.7}$$
$$\text{IMG} = N \times M. \tag{4.8}$$

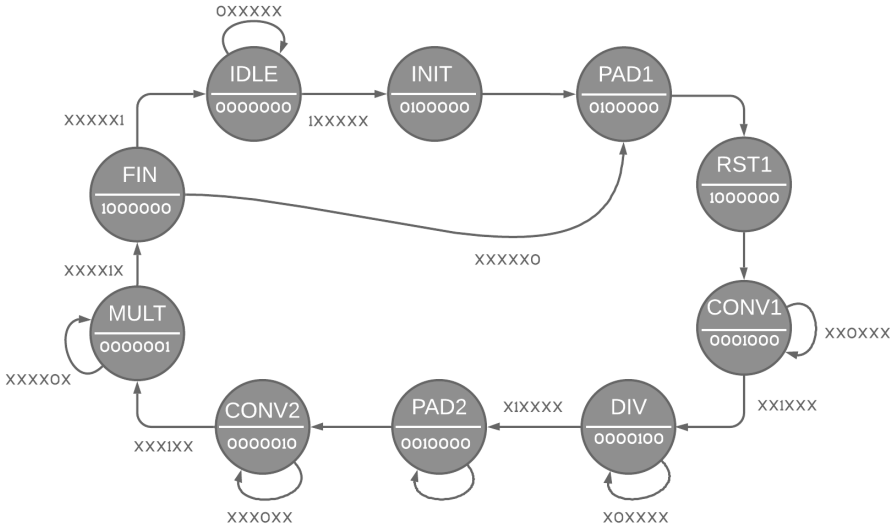


Figure 4.14: State diagram for the master controller for Architecture-2.

Table 4.5: Input/output signal names for the master controller.

Bit	Input	Bit	Output
0	enable_module	0	start_mult
1	s01_axis_tvalid	1	start_conv_2
2	do_div	2	start_div
3	do_mult	3	start_conv_1
4	stop_mult	4	enable_pad_reg_2
5	iter	5	enable_pad_reg
		6	reset_address

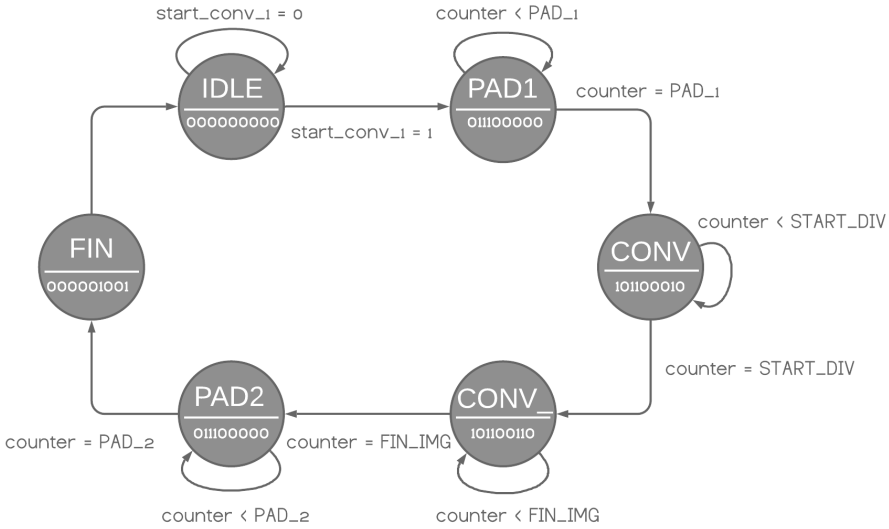


Figure 4.15: State diagram for Convolution1 controller.

Table 4.6: Input/output signal names for the Convolution1 controller.

Bit	Output
0	reset_address
1	read_bram_0
2	do_div
3	stop_conv_1
4	reset_counter
5	enable_counter
6	enable_conv_1
7	enable_pad
8	ena_0

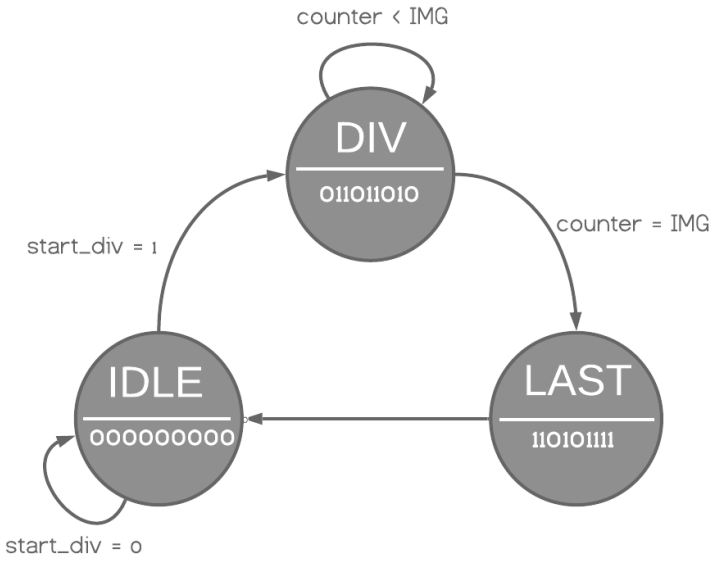


Figure 4.16: State diagram for the Division controller.

Table 4.7: Input/output signal names for the Division controller.

Bit	Output
0	m00_axis_tlast
1	m00_axis_tvalid
2	m01_axis_tlast
3	m01_axis_tvalid
4	s01_axis_tready
5	reset_counter
6	enable_counter
7	enable_div
8	stop_div

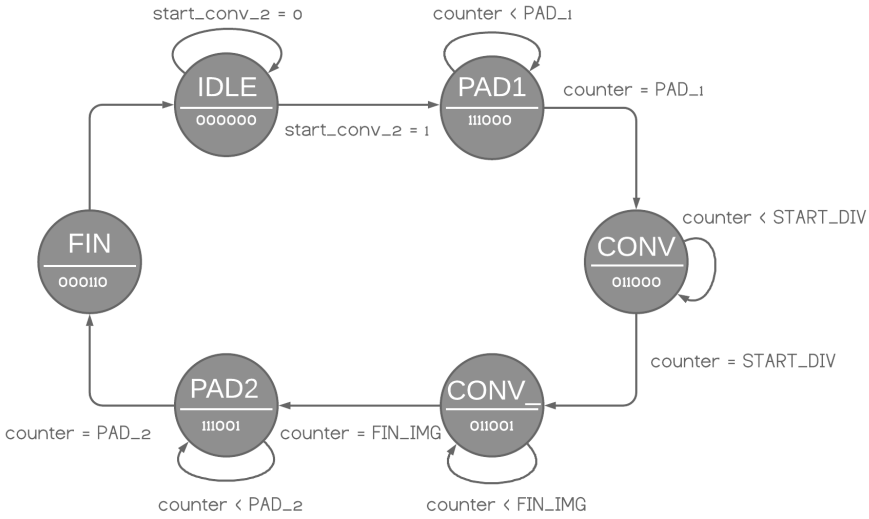


Figure 4.17: State diagram for the Convolution2 controller.

Table 4.8: Input/output signal names for the Convolution2 controller.

Bit	Output
0	`do_mult`
1	`stop_conv_2`
2	`reset_counter`
3	`enable_counter`
4	`enable_conv_2`
5	`enable_pad_2`

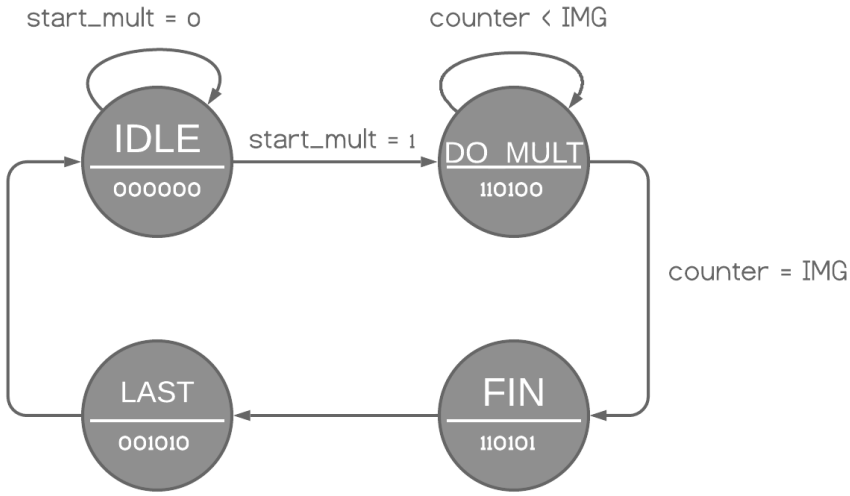


Figure 4.18: State diagram for the Multiplication controller.

Table 4.9: Input/output signal names for the Multiplication controller.

Bit	Output
0	m02_axis_tlast
1	reset_counter
2	enable_counter
3	stop_mult
4	enable_mult
5	enable_read

Chapter 5

Verification

For hardware implementation a Zedboard platform [52] with a built-in Zynq-7020 FPGA is used. The main features of the Zynq-7020 All Programmable SoC are summarized in Table 5.1.

Table 5.1: Feature summary for Zynq-7020 [1].

Processing System		**Programmable Logic**	
Maximum frequency	667 MHz	Maximum frequency	250 MHz
L2 Cache	512 KB	Block RAM (#36 *Kb*)	4.9 Mb (140)
On-Chip Memory	256 KB	DSP Slices	220
		LUTs	53,200
		LUTRAMs	17,400
		FFs	106,400

Both architectures, Architecture-1 and Architecture-2, are implemented in the Xilinx Vivado development environment [53] and described in *VHSIC Hardware Description Language* (VHDL). Architecture-1 and Architecture-2 implementations are tested in simulations. In addition Architecture-1 is tested on a target FPGA, i.e., Zynq-7020.

The chapter is divided into two sections, where Section 5.1 presents verification method and testing results for Architecture-1 and Section 5.2 presents verification method and testing results for Architecture-2.

5.1 Architecture-1

5.1.1 Functional Verification

The functionality of the RL-deconvolution, both the standard- and the accelerated versions, implemented in Architecture-1 is tested against the software implementation written in C. Software is implemented using fixed-point representation and using VC-BC boundary conditions i.e., equivalent to the algorithm implementation in Architecture-1. Verification model consist of two parts, the C-code block and the testbech written in VHDL, shown in Figure 5.1. The C-code block produces the degraded image and a reference restored image, both of which are saved in a corresponding text files. In the tesbench these files are read and stored internally. Before the RL-deconvolution is enabled, the number of iterations is decided and written to the the *Design Under Test* (DUT) through the AXI4-Lite interface. The degraded image is sent to DUT and the restored by the DUT image is compared to the reference restored image in the monitor block shown in Figure 5.1. If the error count is equal to zero, the test is considered to be passed.

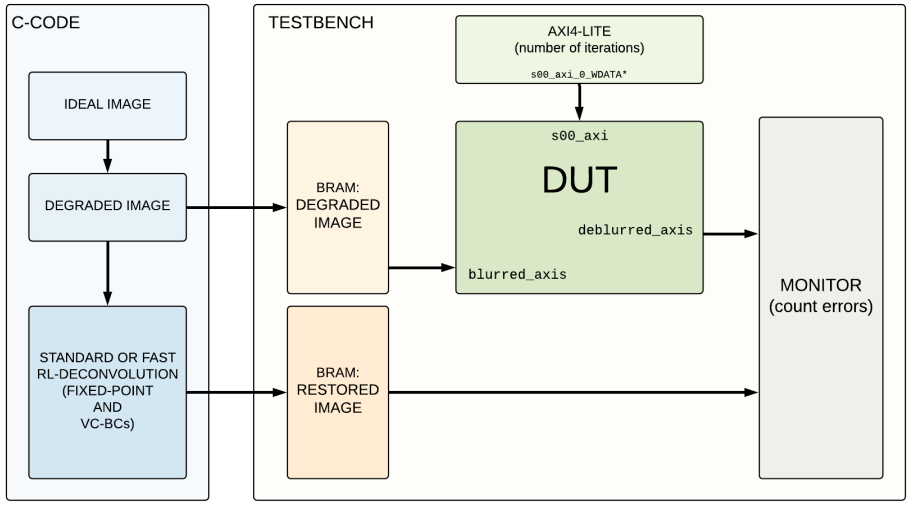


Figure 5.1: Verification of Architecture-1. DUT - design under test.

One band taken from the Urban dataset, shown in Figure 5.2(a), is used for the functional verification. The data is coded with 16 bits. The image is degraded in a software model with a Gaussian kernel with $\sigma = 2.3$, shown in Figure 5.2(b). The degraded image is restored in both the software module and in the implemented hardware module. Both the standard and the accelerated RL-deconvolution are tested. The error between the software generated output, $\hat{\mathbf{X}}^p_{ideal}$, and a hardware generated output, $\hat{\mathbf{X}}^p$, are found by finding the Euclidean distance between the two. Standard and accelerated RL-deconvolution are tested for 1 and 10 iterations. In all cases the error is equal to zero, so both software and hardware modules produce identical results. Visual restoration results for standard RL-deconvolution after 10 iterations are shown in Figure 5.2(a) and Figure 5.2(b), using the software and hardware respectively.

(a) Reference image

(b) Image degraded by the Gaussian PSF with $\sigma = 2.3$.

(c) Restored image using Software. 10 RL-deconvolution iteration.

(d) Restored image using Hardware accelerator. 10 RL-deconvolution iteration.

Figure 5.2: Functional verification for Architecture-1. Visual deconvolution results.

5.1.2 Verification on hardware: interfacing with Zynq 7000 SoC

Architecture-1 is connected to the Processing system and further tested on a Zedboard platform. The whole system consist of an ARM Cortex-A9 CPU in the processing side, an external memory, *DDR3*, a direct memory access, *DMA*, IP, a concat IP core and a *RL-Deconvolution* IP. A generalized block design is shown in Figure 5.3. DMA is initialized by the PS through the general purpose, *GP0*, port using the AXI4-Lite interface. When initialized, DMA gets the master access to the DDR through the HP0 port. The communication between the DMA and the DDR happens using the AXI4-memory mapped interface. With the custom RL-deconvolution Module DMA communicates via the AXI4-stream interface. RL-Deconvolution module is also initialized by the PS through the same HP0 port. An additional AXI Interconnect IP ensures the correct routing of the signal from the GP0 port to the corresponding modules. Concat IP core [54] is used to combine several bus signals into one. In this design, concat IP core is used to connect the DMA interrupts to the PS's interrupt request, *IRQ*, generator. DMA has two interrupt outputs, one for the Memory Map to Stream channel, `mm2s_introut` and one for the Stream to Memory Map channel, `s2mm_introut`. The PS gets an interrupt when either the last element of the input image is send to the DMA and the `m_axis_mm2s_wlast` signal is set high or when the last element of the output image is send to the PS and the `m_axi_s2mm_rlast` signal is set to one. The PS will get notified by the interrupt system when the RL-deconvolution is finished processing data.

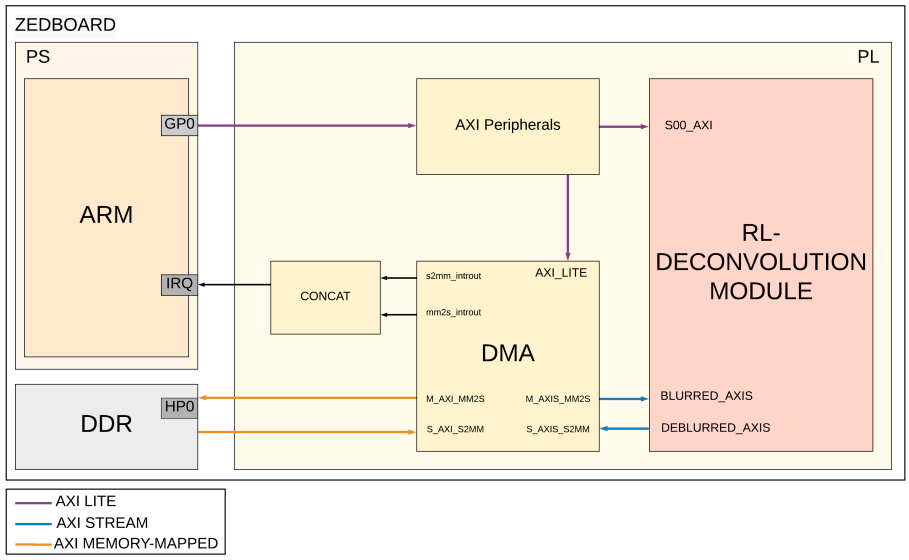


Figure 5.3: The overall architecture for RL-deconvolution.

The generated bit-stream is exported to the Xilinx *Software Development Kit* (SDK), from which the implemented hardware design is tested. Processing is done band by band for a 3-D hyperspectral image. A C-script is responsible for reading one image band from an SD-card, storing these band in the external memory and initializing the hardware module. The initialization starts by writing the desired number of iterations, k, into the configuration register, and also deciding which version of the algorithm to run. The degraded band, $\mathbf{Y}^p$, is sent to the hardware module and after k iterations, the restored image $\hat{\mathbf{X}}^p$ is received by the PS and is saved in the SD-card. The process repeats for all bands in the hyperspectral data set. The flow of the RL-deconvolution for hyperspectral images is shown in Figure 5.4.

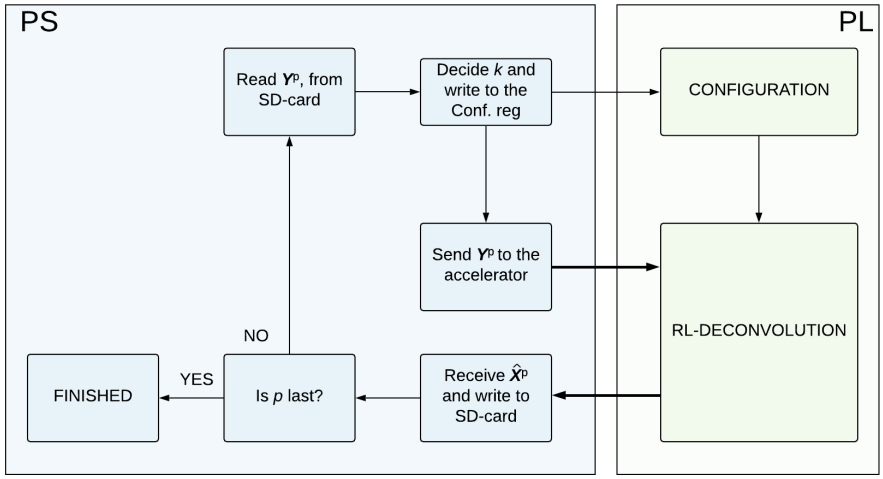


Figure 5.4: Flow of the RL-deconvolution for hyperspectral images.

5.2 Architecture-2

5.2.1 Functional and RTL Verification

The functionality of the RL-deconvolution, both the standard- and the accelerated versions, implemented in Architecture-2 is tested against the software implementation written in C. Software is implemented using fixed-point representation, and boundary conditions equal to MP-BCs, i.e., equivalent to the algorithm implementation in Architecture-1. The complete testbech is shown in Figure 5.5. In consists of a C-code block, which produces the degraded image and a reference restored image, and a testbech written in VHDL. In the testbech the degraded image and the reference restored image are read from a text file and stored in the BRAMs. Then, the number of iterations and σ for the restoration kernel are decided and written to the RL-deconvolution module through the AXI4-Lite interface. The degraded image is sent to DUT and the restored by DUT image is compared to the reference restored image by the monitor block shown in Figure 5.5. If the error count is equal to zero, the test is considered to be passed.

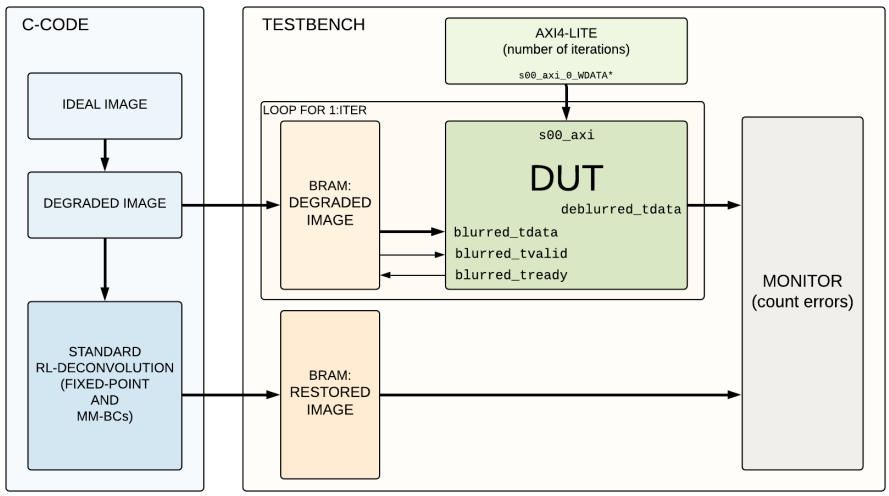
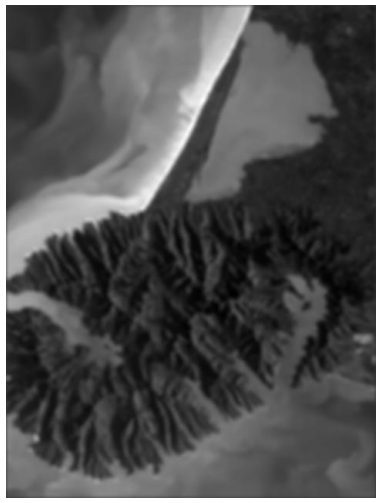


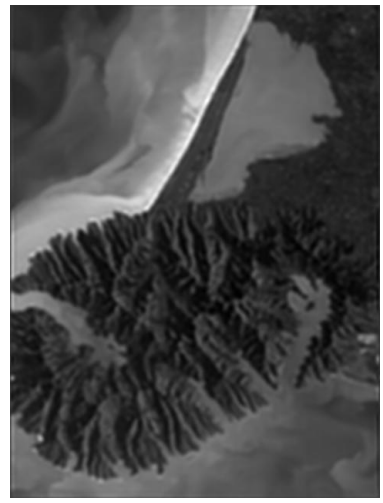
Figure 5.5: Verification of Architecture-2.

The difference between the testbench for Architecture-2 and Architecture-1 is a loop block around BRAM storing the degraded image and DUT, shown in Figure 5.5. The degraded image needs to be send to the RL-deconvolution module for each new iteration. The image is fetched from the BRAM when the signal `blurred_axis_tready` is set to 1 by the DUT.

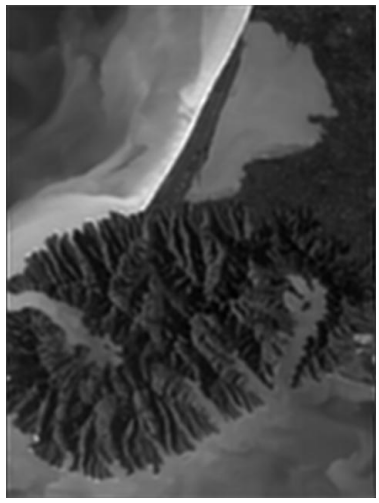
For simulation purposes one spectral band from the Coast hyperspectral data set is degraded in the software module with a Gaussian kernel of size 7×7 and 9×9, the result with a kernel of size 9×9 is shown in Figure 5.6(a). Then, the images are restored in both the software and hardware module. The restoration kernels are equal to the corresponding degradation kernels. The deconvolution result after 10 RL-deconvolution iterations using the kernel of size 9×9 is shown in Figure 5.6(b) and Figure 5.6(c), for software and hardware respectively. The difference between the software generated output, $\hat{\mathbf{X}}^p_{true}$, and hardware generated output, $\hat{\mathbf{X}}^p$, is found by calculating the Euclidean distance between the two. The error matrix is shown in Figure 5.6(d). The error is equal to zero, except for the upper and bottom rows. This is due to the way convolution is done in the Architecture-2. This is illustrated in Figure 2.3 in Section 2.1, where the first and the last output samples are generated randomly. The amount of this random samples depends on the kernel size, e.g., for a kernel of size 9×9, the number is equal to 4 samples. These four samples are different in the software and hardware implementations. For each iterations the difference grows due to the way convolution is performed. This is the expected behaviour, and does not affect the deconvolution results.



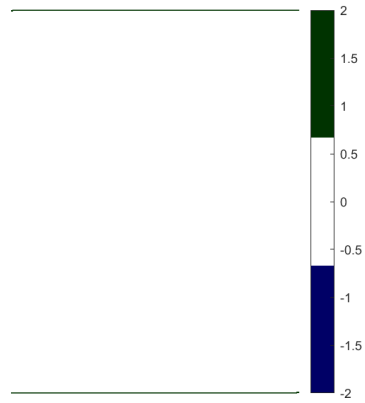
(a) Degraded image.



(b) Software: 10 iter.



(c) Hardware: 10 iter.



(d) Error matrix (size equal to the image size).

Figure 5.6: Visual restoration results from the functional verification for Architecture-2.

Chapter 6

Results and Discussion

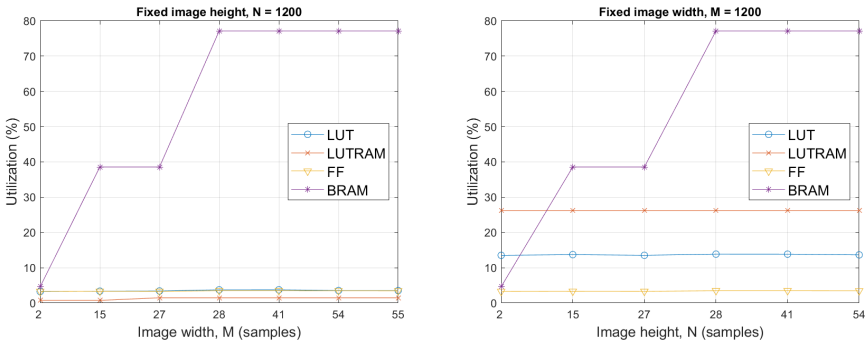
This chapter gives the results, regarding timing and resource utilization, for the Architecture-1 and Architecture-2 implementations on a target FPGA, i.e., Xilinx Zynq-7020.

6.1 Resource utilization

6.1.1 Architecture-1

Architecture-1 uses three BRAM modules, where two of them ($BRAM\_G$ and $BRAM\_F$) store the data with width equal to 16 bits and one ($BRAM\_T$) store the data with width equal to 22 bits. As explained in Appendix C, the maximum number of elements which can be stored in each BRAMs is equal to 84954. For an image of size 1200×54, the total amount of block RAMs needed for storing the images is equal to 108. Figure 6.1 shows the resource utilization for the Architecture-1 using a variable image size $N \times M$. Figure 6.1(a) shows the utilization results as a function of image width, when image height N is equal to 1200. Figure 6.1(b) shows the utilization results as a function of image height, when image width M is equal to 1200.

Architecture-1 uses 20 DSPs, not shown in Figure 6.1, independent of the image size, where 18 DSPs are used in the convolution module and 2 DSPs are used in the multiplication module. Multiplication uses two DSPs due to the implementation of the accelerated RL-deconvolution. The line buffers in the convolution module are implemented in LUTRAMs as distributed RAM. The size of the line buffer depends on the image width. This can be seen from Figure 6.1(b), where the resource usage in respect to the LUTRAMs, as well as FFs and LUTs, increases



(a) **(b)**

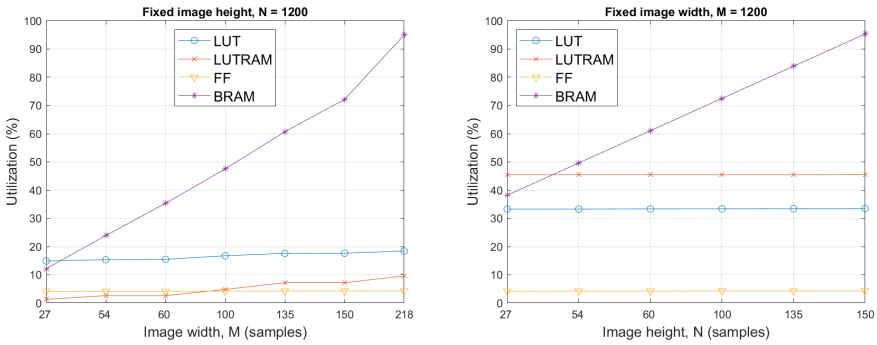
Figure 6.1: Architecture-1, resource utilization as a function of the image width (a) and as a function of the image height (b).

when the width is set to 1200, i.e., $M > N$.

6.1.2 Architecture-2

Architecture-2 stores one image of size $N \times M$ with width equal to 16 bits in BRAMs. In addition, there are two FIFOs implemented in BRAMs storing data with width equal to 16 bits and 32 bits. If the width of the image is set to a fixed constant, then the maximum image height can be found as described in Appendix C, e.g., for an image with width equal to $M = 1200$, the maximum height is equal to $N = 150$. On the other hand, if the image height is fixed, then for an image with height equal to 1200, the maximum width is then equal to 218. Figure 6.2(a) shows utilization results as a function of image width, when image height N is equal to 1200. Figure 6.2(b) shows the utilization results as a function of image height, when image width M is equal to 1200.

Architecture-2 uses 45 DSPs independent of the image size, where 2×18 DSPs are used in convolution modules and 2 DSPs are used in the multiplication module. The rest of the DSPs are used in the controller module. As seen in Figure 6.2, the resource usage is higher if the image width is bigger than the image height. The reason is the use of the image width dependent line buffers in the convolution module, similar to the Architecture-1.



(a) **(b)**

Figure 6.2: Architecture-2, resource utilization as a function of the image width (a) and as a function of the image height (b).

6.2 Execution time

6.2.1 Architecture-1

For an image of size 1200×54 the *Worst Negative Slack* (WNS) is equal to 2.0 ns when the frequency is set to 100 MHz. This means that the maximum frequency of 125 MHz can be achieved. The execution time for a bare-metal application run on the Xilinx Zynq SoC is measured using the Xilinx library *xtime_l.h*. The global timer is used, where a counter is increased every two clock cycles. Image is of size 260×250. The time measured from the time the last valid input pixel is sent to the accelerator to the time the last valid output pixel is received. One RL-deconvolution iteration takes 138358 clock cycles or 1.38 ms. The initial latency is equal to $N \times M = 65000$ clock cycles. Execution time is plotted as a function of number of iterations, without considering the initial latency, shown in Figure 6.4. The plot is the same for both the standard and the accelerated RL-deconvolution. The difference in the number of iterations needed in the case of the standard and accelerated RL-deconvolution is shown in Figure 6.3.

In Figure 6.4 the M-RRE is shown for both algorithms, where the same M-RRE is achieved twice as fast for the accelerated RL-deconvolution compared to the standard RL-deconvolution.

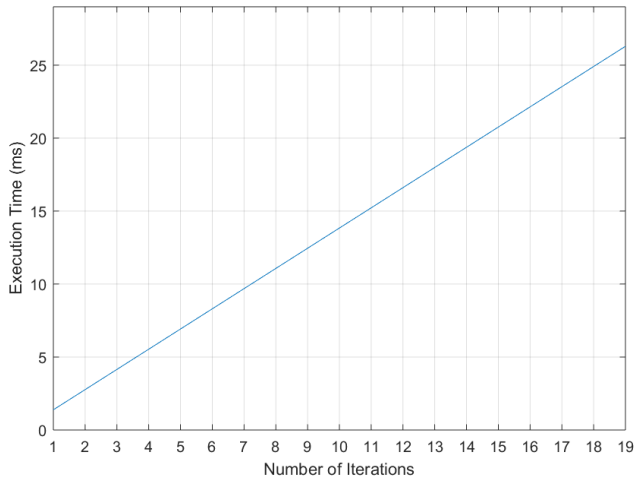


Figure 6.3: Execution time as a function of number of iterations. Measured from the time the last input element is sent until the last element of the output image is received.

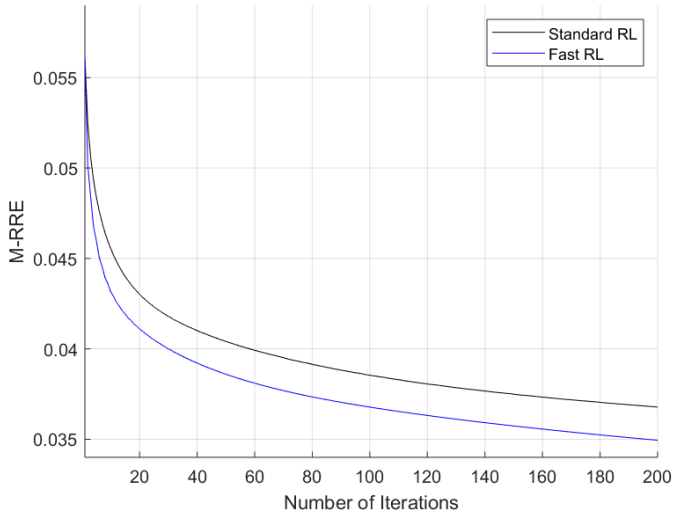


Figure 6.4: M-RRE as a function of number of iterations.

6.2.2 Architecture-2

There are two slightly different versions of Architecture-2, the one is run-time configurable with respect to the kernel size and another one has a fixed kernel size. The maximum frequency of the former version is equal to 50 MHz, while the maximum frequency of the latter is equal to 136 MHz. The implementation difference does not affect the total number of clock cycles. Execution time for Architecture-2 is taken from the simulations. The tested image is of size 260×250. One iteration takes 69071 clock cycles (or 0.69 ms running on 100 MHz).

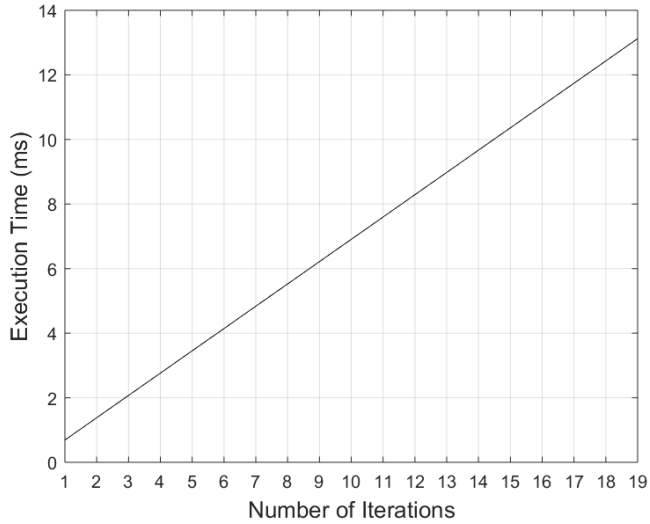


Figure 6.5: Execution time as a function of number of iterations for Architecture-2.

6.3 Power estimation

6.3.1 Architecture-1

Power estimation are done on the post-synthesis design using the power estimation tools provided bu Xilinx Vivado. Figure 6.6 shows power estimation when using an image of size equal to either $1200 \times M$, marked red, or $N \times 1200$, marked blue.

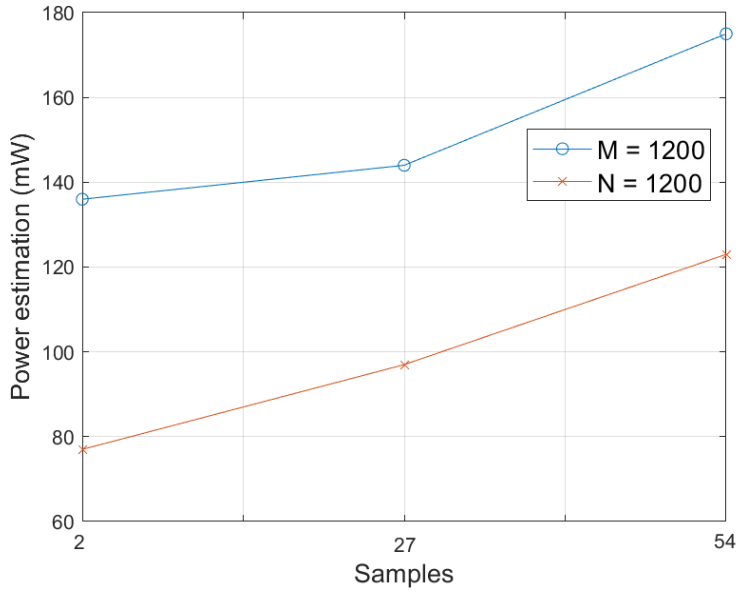


Figure 6.6: Power estimation for Architecture-1.

As for the resource utilization, the power usage is higher for $M > N$.

6.4 Discussion

Resource usage

As seen in Section 6.1, for both Architecture-1 and Architecture-2, the resource utilization depends on the image orientation, i.e., the resource usage is higher if image width is larger than the image height. It should be noted, that the usage of BRAMs depends solely on the total amount of stored samples. For an image size equal to 1200×54, which is maximum possible image size for the Architecture-1 implemented on Zynq-7020, $\sim$ 80 % of BRAMs are used in Architecture-1 and $\sim$ 20% of BRAMs are used in Architecture-2. The result is that the Architecture-2 can work with image containing 196800 more samples. Compared to other resources, the Architecture-1 uses $\sim$ 5 % of available LUTs, while the Architecture-2 uses $\sim$ 15 % of available LUTs. The biggest disadvantage of the Architecture-2, is a higher DSP element use, that is twice as high as for the Architecture-1.

Execution time

Architecture-1 and Architecture-2 are compared, with respect to the execution time, to the state-of-the-art implementations presented in Section 2.4.3, the HW/SW codesign implementation of RL-deconvolution algorithm presented in Section 2.4.2 and a software only implementation of RL-deconvolution tested on the target FPGA. Table 6.1 shows the comparison between the different implementations.

Table 6.1: Comparison between several RL-deconvolution implementations.

	Iter.	Image size	PSF	Time (ms)	Freq. (MHz)
[47]	15	640×480	<10	40	-
[45]	10	800×525	9×9	80	61
[44]	60	64×64	13×13	78	100
SW-Only	1	512×512	9×9	164	100
HW/SW codesign	1	512×512	9×9	70.1	100
Proposed Arch-1	1	640×480	9×9	6.32	100
Proposed Arch-2[1]	1	640×480	9×9	3.14	100

[1]Execution time is shown for the Architecture-2 which does not support the run-time configurable kernel size.

Assuming, for an easier comparison, that the execution time scales linearly and that each iteration takes equal amount of time. The estimated execution times are shown in Table 6.2

Table 6.2: Comparison between several RL-deconvolution implementations.

	Iter.	Image size	PSF	Time (ms)	Freq. (MHz)
[47]	1	640×480	<10	2.66	-
[45]	1	640×480	9×9	5.85	61
[44]	1	640×480	13×13	97.5	100
SW-Only	1	640×480	9×9	192.19	100
HW/SW codesign	1	640×480	9×9	82.14	100
Proposed Arch-1	1	640×480	9×9	6.32	100
Proposed Arch-2	1	640×480	9×9	3.14	100

From Table 6.2, it is clear that both proposes architectures outperforms the previously implemented SW-only and solution and HW/SW codesign solutions. The speed-up by a factor of 30.4 and 13.0 is achieved for the Architecture-1 compared to the SW-Only implementation and HW/SW codesign implementation, respectively and the speed-up by a factor of 61.2 and 26.2 is achieved for the Architecture-2 compared to the SW-Only implementation and HW/SW codesign implementation, respectively. Both proposed architectures do also compare well with the state-of-the-art solutions. The architecture in [45] is most similar, in terms of the algorithm implementation (i.e., kernel is assumed to be space-invariant and the images are extended before performing the convolution), to the proposed architectures. A speed-up by a factor of 1.8 is achieved when comparing the Architecture-2 to [45]. In addition, the proposed architectures implement the accelerated RL-deconvolution, with $\beta = 2$, thus decreasing a number of iterations to be executed by a factor of two.

Chapter 7

Conclusion

RL-deconvolution algorithm, used for reducing degradation (e.g., the optical blur) in hyperspectral images, was designed and implemented in FPGA. The 3-D hyperspectral images were assumed to be cross-channel independent, and were therefore modeled as a collection of 2-D independent images. Degradation was modeled as a convolution between the 2-D images and a point spread function, also called kernel, was assumed to be estimated prior to the deconvolution. The chosen algorithm has slow convergence rate, hence an accelerated RL-deconvolution version yielding a decrement in the number of iterations to be executed was implemented alongside the standard RL-deconvolution. It was found that the standard RL-deconvolution, in some cases (e.g., in the presence of noise), gave better results than the accelerated RL-deconvolution, thus it was decided to have a possibility to choose the algorithm version at a run-time. The input image size was preserved by extending the image boundaries before performing convolution. It was found that the boundary conditions, called VC-BCs, performs best compared to the two other tested BCs, i.e., Z-BCs and MP-BCs. The comparison was done with respect to the M-PSNR, M-SSIM and by visual comparison.

Two architectures were implemented, one (called Architecture-1) optimized with respect to the communication with the external memory and the other one (called Architecture-2) optimized with respect to the internal storage. Both implemented architecture are scalable with respect to the image size and run-time configurable with respect to the number of RL-deconvolution iterations. This results in a more general solution compared to some of the state-of-the-art implementations, which support only a limited amount of iterations. In addition, the Architecture-2 is run-time configurable with respect to the kernel size, where maximum kernel size is equal to 9×9. The use of a correct kernel size is important as a wrongly chosen one can worsen the degraded image. Therefore a flexible design in term of the kernel size is significant.

The proposed hardware implementations are compared to a Software-Only implementation of the RL-deconvolution running on a target platform (i.e., Xilinx Zynq-7020) and a HW/SW codesign for RL-deconvolution, where the time consuming convolution is accelerated in FPGA. A speedup by a factor of 30 and 60 is achieved compared to a Software-Only for Architecture-1 and Architecture-2, respectively. A speedup by a factor of 13 and 26 is achieved compared to a HW/SW codesign implementations for Architecture-1 and Architecture-2, respectively. The calculated speed-up does not take the communication with an external memory into the account. Compared to the state-of-the-art implementation of the RL-deconvolution, a speed-up by a factor of 1.8 is achieved for Architecture-2.

7.1 Future Work

The Architecture-2 was optimized with respect to the internal storage on FPGA, nevertheless the proposed architecture still uses a high amount of BRAM components. The number depends on the degraded image size. A possible future work would be to store all the data in the external memory. The increase in the communication with the external memory could be a possible bottleneck. In addition, the run-time configurable with respect to the kernel size version of the Architecture-2 has a maximum frequency equal to 50 MHz. This could possibly be improved by further pipelining the design.

In addition, a few extra features could be implemented; image registration, kernel estimation and a stopping criteria for the RL-deconvolution algorithm. These features together with the proposed deconvolution implementation would make a complete image spatial reconstruction system ready to be placed on a satellite.

Bibliography

[1] X. Inc., "Zynq-7000 all programmable soc data sheet: Overview," 2017.

[2] NASA Ocean Biology Processing Group, "Hico level 1 data version 1,"

[3] X. Inc., "AXI reference guide." https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, 2011. [Online; accessed 27-May-2019].

[4] F. Zhu, Y. Wang, B. Fan, G. Meng, and C. Pan, "Effective spectral unmixing via robust representation and learning-based sparsity," *CoRR*, vol. abs/1409.0685, 2014.

[5] X. Inc., "Divider generator (v5.1)." https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf, 2016. [Online; accessed 27-May-2019].

[6] Z. Zhang and J. C. Moore, "Chapter 4 - remote sensing," in *Mathematical and Physical Fundamentals of Climate Change* (Z. Zhang and J. C. Moore, eds.), pp. 111 – 124, Boston: Elsevier, 2015.

[7] The National Center for Water Quality Research, "Algae vs. "harmful algae" – what's the difference?." http://lakeeriealgae.com/algae-vs-harmful-algae/, 2015. [Online; accessed 3-March-2019].

[8] R. B. Smith, *Introduction to Hyperspectral Imaging*. MicroImages, 2006.

[9] Harris Geospatial Solutions, Inc., "Radiance vs. reflectance." https://www.harrisgeospatial.com/Support/Self-Help-Tools/Help-Articles/Help-Articles-Detail/ArtMID/10220/ArticleID/19247/3377, 2019. [Online; accessed 19-June-2019].

[10] C. O. Davis and J. Nahorniak, "Hico on-line atmospheric correction." https://hyspiri.jpl.nasa.gov/downloads/2015_Symposium/day3/9_HICOOn-LineAtmosphericCorrection_Davis.pdf, 2015. [Online; accessed 3-March-2019].

[11] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[12] F. Sigernes, D. A. Lorentzen, K. Heia, and T. Svenøe, "Multipurpose spectral imager," *Applied Optics*, vol. 39, p. 3143, June 2000.

[13] J. Fjeldtvedt and M. Orlandić, "Cubedma – optimizing three-dimensional dma transfers for hyperspectral imaging applications," *Microprocessors and Microsystems*, vol. 65, pp. 23 – 36, 2019.

[14] M. Orlandić, S. Bakken, and T. A. Johansen, "The effect of dimensionality reduction on signature-based target detection for hyperspectral imaging," 2019.

[15] M. Orlandić, D. Bošković, and T. A. Johansen, "HW/SW Implementation of hyperspectral target detection algorithm," 2019.

[16] J. Fjeldtvedt, M. Orlandić, and T. A. Johansen, "An efficient real-time fpga implementation of the ccsds-123 compression standard for hyperspectral images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, pp. 3841–3852, Oct 2018.

[17] M. Orlandić, J. Fjeldtvedt, and T. A. Johansen, "A parallel fpga implementation of the ccsds-123 compression algorithm," *Remote Sensing*, vol. 11, no. 6, 2019.

[18] Sung Cheol Park, Min Kyu Park, and Moon Gi Kang, "Super-resolution image reconstruction: a technical overview," *IEEE Signal Processing Magazine*, vol. 20, pp. 21–36, May 2003.

[19] E. M. Grotte, B. R., J. F. Fortuna, J. Veisdal, O. M., and H.-L. E., "Hyperspectral imaging small satellite in multi-agent marine observation system," 2017.

[20] M. Bertero, P. Boccacci, G. Desidera, and G. Vicidomini, "Image deblurring with poisson data: From cells to galaxies," *Inverse Problems*, vol. 25, p. 123006, 11 2009.

[21] S. Henrot, C. Soussen, and D. Brie, "Fast positive deconvolution of hyperspectral images," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 22, 09 2012.

[22] R. A. Schowengerdt, *Remote Sensing, Third Edition: Models and Methods for Image Processing*. Orlando, FL, USA: Academic Press, Inc., 2006.

[23] M. Almeida and M. Figueiredo, "Deconvolving images with unknown boundaries using the alternating direction method of multipliers," *Image Processing, IEEE Transactions on*, vol. 22, pp. 3074–3086, 08 2013.

[24] X. Zhou, F. Zhou, X. Bai, and B. Xue, "A boundary condition based deconvolution framework for image deblurring," *Journal of Computational and Applied Mathematics*, vol. 261, p. 14–29, 05 2014.

[25] R. C. Gonzalez and R. E. Woods, "Digital image processing," 2002.

[26] K. Dines and A. Kak, "Constrained least squares filtering," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, pp. 346–350, August 1977.

[27] B. Hunt and O. Kubler, "Karhunen-loeve multispectral image restoration, part i: Theory," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 32, pp. 592 – 600, 07 1984.

[28] N. Galatsanos and R. Chin, "Digital restoration of multi-channel images," in *ICASSP '87. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 12, pp. 1244–1247, April 1987.

[29] N. P. Galatsanos, A. K. Katsaggelos, R. T. Chin, and A. D. Hillery, "Least squares restoration of multichannel images," *IEEE Transactions on Signal Processing*, vol. 39, pp. 2222–2236, Oct 1991.

[30] B. R. Hunt, "Super-resolution of images: Algorithms, principles, performance," *International Journal of Imaging Systems and Technology*, vol. 6, pp. 297 – 304, 10 2005.

[31] W. H. Richardson, "Bayesian-based iterative method of image restoration∗," *J. Opt. Soc. Am.*, vol. 62, pp. 55–59, Jan 1972.

[32] L. B. Lucy, "An iterative technique for the rectification of observed distributions," *Astron. J.*, vol. 79, pp. 745–754, 1974.

[33] L. A. Shepp and Y. Vardi, "Maximum likelihood reconstruction for emission tomography," *IEEE Transactions on Medical Imaging*, vol. 1, pp. 113–122, Oct 1982.

[34] H. Lanteri, M. Roche, O. Cuevas, and C. Aime, "A general method to devise maximum-likelihood signal restoration multiplicative algorithms with non-negativity constraints," *Signal Processing*, vol. 81, pp. 945–974, 05 2001.

[35] David S. C. Biggs and M. Andrews, "Acceleration of iterative image restoration algorithms," *Appl. Opt.*, vol. 36, pp. 1766–1775, Mar 1997.

[36] M. Prato, R. Cavicchioli, L. Zanni, P. Boccacci, and M. Bertero, "Efficient deconvolution methods for astronomical imaging: Algorithms and idl-gpu codes," *Astronomy and Astrophysics*, vol. 539, p. A133, 03 2012.

[37] E. S. Meinel, "Origins of linear and nonlinear recursive restoration algorithms," *J. Opt. Soc. Am. A*, vol. 3, pp. 787–799, Jun 1986.

[38] M. Bertero and P. Boccacci, "A simple method for the reduction of boundary effects in the richardson-lucy approach to image deconvolution," *http://dx.doi.org/10.1051/0004-6361:20052717*, vol. 437, 07 2005.

[39] R. L. Lagendijk, J. Biemond, and D. E. Boekee, "Regularized iterative image restoration with ringing reduction," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 1874–1888, Dec 1988.

[40] H. Fang, C. Luo, G. Zhou, and X. Wang, "Hyperspectral image deconvolution with a spectral-spatial total variation regularization," *Canadian Journal of Remote Sensing*, 08 2017.

[41] J. Jemec, F. Pernuš, B. Likar, and M. Bürmen, "Deconvolution-based restoration of swir pushbroom imaging spectrometer images," *Opt. Express*, vol. 24, pp. 24704–24718, Oct 2016.

[42] K. Avagian, "Lucy-Richardson deconvolution of hyper-spectral images on FPGA," tech. rep., Norwegian University of Science and Technology, 2018. Semester project.

[43] M. B. P. Gege and S. Holzwarth, "Aiborne remote sensing, rosis bonn 2008 summary report," 2008.

[44] Z. Wang, K. Weng, Z. Cheng, L. Yan, and J. Guan, "A co-design method for parallel image processing accelerator based on dsp and fpga," vol. 8005, pp. 8005 – 8005 – 6, 2011.

[45] O. Anacona-Mosquera, J. Arias-García, D. M. Muñoz, and C. H. Llanos, "Efficient hardware implementation of the richardson-lucy algorithm for restoring motion-blurred image on reconfigurable digital system," in *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pp. 1–6, Aug 2016.

[46] O. Sims, "Efficient implementation of video processing algorithms on fpga," 2007.

[47] S. Carrato, G. Ramponi, S. Marsi, M. Jerian, and L. Tenze, "Fpga implementation of the lucy-richardson algorithm for fast space-variant image deconvolution," in *2015 9th International Symposium on Image and Signal Processing and Analysis (ISPA)*, pp. 137–142, Sept 2015.

[48] The Mathworks, Inc., Natick, Massachusetts, *MATLAB version R2018a*, 2018.

[49] X. Inc., "7 series FPGAs memory resources." `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`, 2017. [Online; accessed 20-June-2019].

[50] M. Orlandić and K. Svarstad, "An adaptive high-throughput edge detection filtering system using dynamic partial reconfiguration," *Journal of Real-Time Image Processing*, Feb 2018.

[51] X. Inc., "FIFO Generator (v13.1)." https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_1/pg057-fifo-generator.pdf, 2017. [Online; accessed 27-May-2019].

[52] ZedBoard, "Zedboard product page." http://zedboard.org/product/. [Online; accessed 9-May-2019].

[53] X. Inc., "Vivado design suite - hlx editions." https://www.xilinx.com/products/design-tools/vivado.html, 2015. [Online; accessed 27-May-2019].

[54] X. Inc., "LogiCORE IP Concat (v2.1)." https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlconcat/v2_1/pb041-xilinx-com-ip-xlconcat.pdf, 2016. [Online; accessed 27-May-2019].

Appendix A

State Diagrams

A.1 Architecture-1

Convolution&Division Controller

Convolution&Division Controller in Architecture-1 is divided into three smaller blocks, the main Convolution&Division module, a Convolution 1 module and a Division module. The state diagram for the Convolution&Division Controller is shown in Figure A.1.1 and the corresponding input/output signal names are shown in Table A.1.1. The state diagram for the Convolution 1 Controller is shown in Figure A.1.2 and the corresponding input/output signal names are shown in Table A.1.2. The state diagram for the Division Controller is shown in Figure A.1.3 and the corresponding input/output signal names are shown in Table A.1.3.

The constants in Figure A.1.1, Figure A.1.2 are equal to

$$\text{START\_DIV} = (M + (W-1)) \times (W-1) + 16 \tag{1.1}$$

$$\text{PAD\_1} = w - 1 \tag{1.2}$$

$$\text{PAD\_2} = (M + (W-1)) \times (W-1)/2 + 4 \tag{1.3}$$

$$\text{STOP\_1} = (N + (W-1)/2) \times (M + (W-1)) - 4 \tag{1.4}$$

$$\text{STOP\_2} = (N + (W-1)) \times (M + (W-1)) \tag{1.5}$$

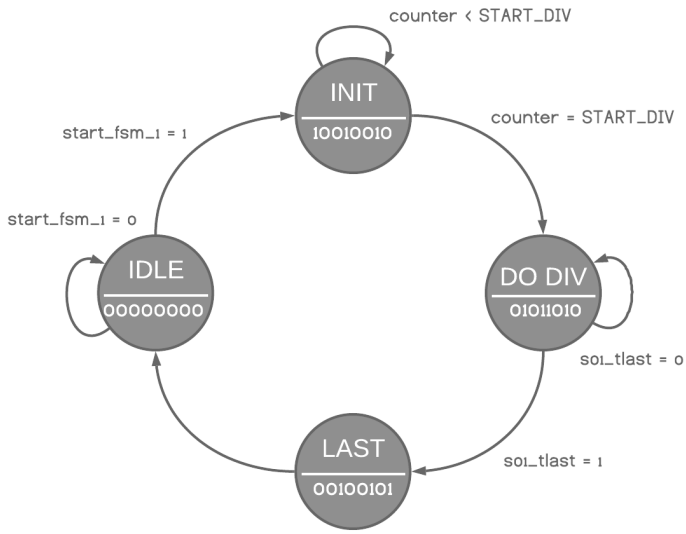


Figure A.1.1: State diagram for the Convolution&Division Controller.

Table A.1.1: Input/Output signal names for the state diagram of Convolution&Division Controller.

Bit	Output
0	`enable_counter_1`
1	`stop_fsm_1`
2	`enable_input_to_convolution`
3	`enable_input_to_division`
4	`reset_counter`
5	`enable_counter_2`
6	`reset_modules`

Convolution 1 Controller

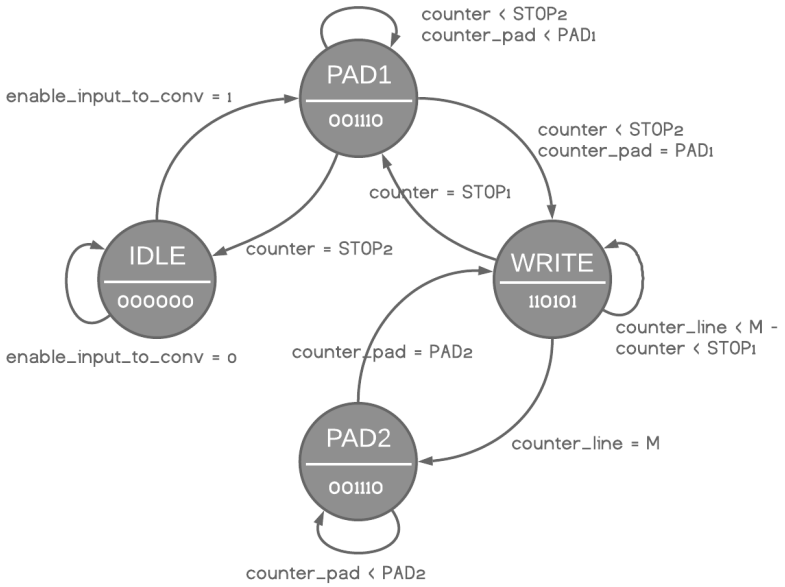


Figure A.1.2: State diagram for the Convolution 1 Controller.

Table A.1.2: Input/Output signal names for the state diagram of Convolution 1 Controller.

Bit	Output
0	`enb_0_division`
1	`enb_0`
2	`enable_pad`
3	`en_conv`
4	`count_pad_en_in`
5	`count_lines_in_en`

Division Controller

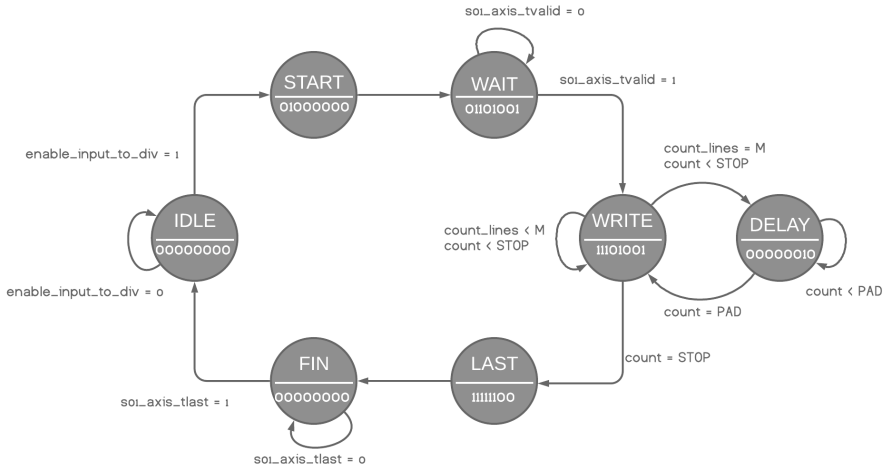


Figure A.1.3: State diagram for the Division Controller.

Table A.1.3: Input/Output signal names for the state diagram of Division Controller.

Bit	Output
0	enable_div
1	enb_1
2	m00_axis_tvalid
3	m00_axis_tlast
4	m01_axis_tvalid
5	m01_axis_tlast
6	count_pad_en_out
7	count_lines_out_en

Convolution&Multiplication Controller

Convolution&Multiplication Controller in Architecture-1 is divided into three smaller blocks as well, the main Convolution&Multiplication module, a Convolution 2 module and a Multiplication module. The state diagram for the Convolution&Multiplication Controller is shown in Figure A.1.4 and the corresponding input/output signal names are shown in Table A.1.4. The state diagram for the Convolution 2 Controller is shown in Figure A.1.5 and the corresponding input/output signal names are shown in Table A.1.5. The state diagram for the Multiplication Controller is shown in Figure A.1.6 and the corresponding input/output signal names are shown in Table A.1.6.

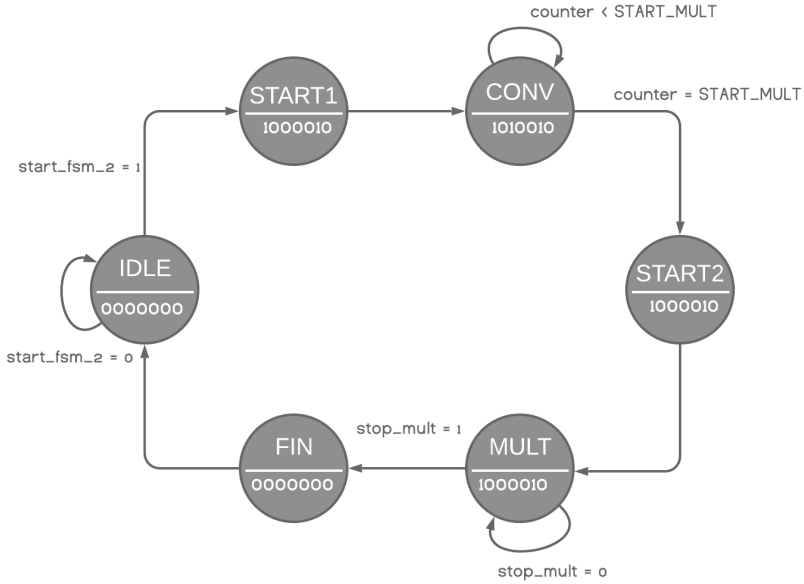


Figure A.1.4: State diagram for the Convolution&Multiplication Controller.

Table A.1.4: Input/Output signal names for the state diagram of Convolution&Multiplication Controller.

Bit	Output
0	`enable_counter_1`
1	`stop_fsm_2`
2	`enable_input_conv`
3	`enable_input_mult`
4	`reset_counter`
5	`enable_counter_2`
6	`reset_modules`

Convolution 2 Controller

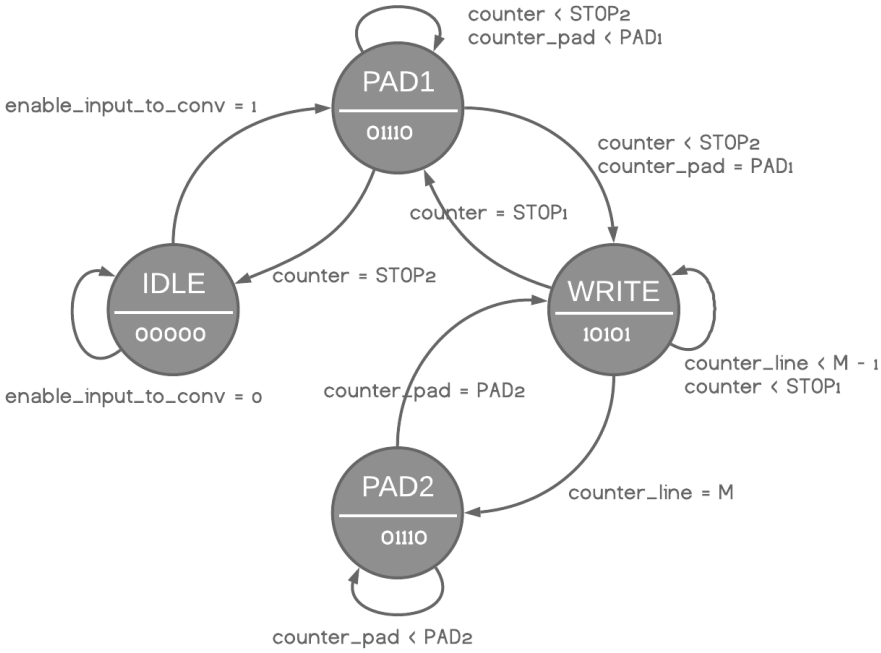


Figure A.1.5: State diagram for the Convolution 2 Controller.

Table A.1.5: Input/Output signal names for the state diagram of Convolution 2 Controller.

Bit	Output
0	`ena_2`
1	`enable_pad`
2	`enable_conv`
3	`count_pad_en_in`
4	`count_lines_en_in`

Multiplication Controller

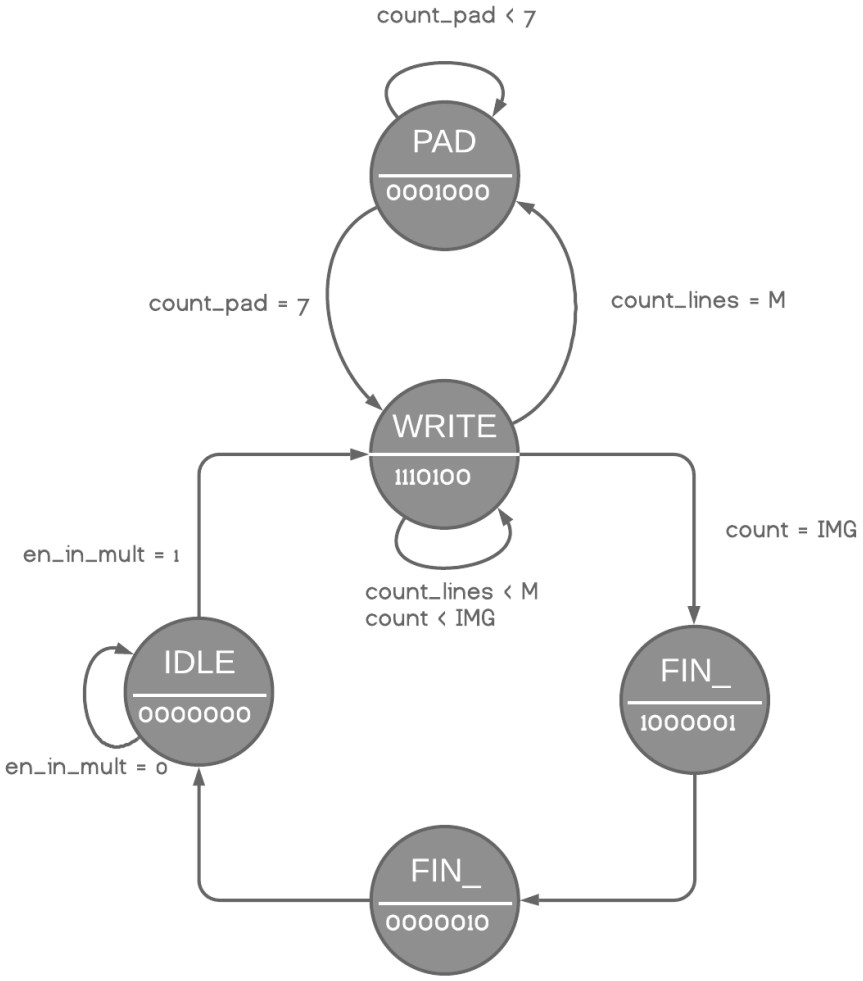


Figure A.1.6: State diagram for the Multiplication Controller.

Table A.1.6: Input/Output signal names for the state diagram of Multiplication Controller.

Bit	Output
0	`enable_write`
1	`en_mult`
2	`enb_0`
3	`count_pad_en_out`
4	`count_lines_en_out`
5	`stop_mult`
6	`last`

Appendix B

Block Diagrams

B.1 Architecture-1

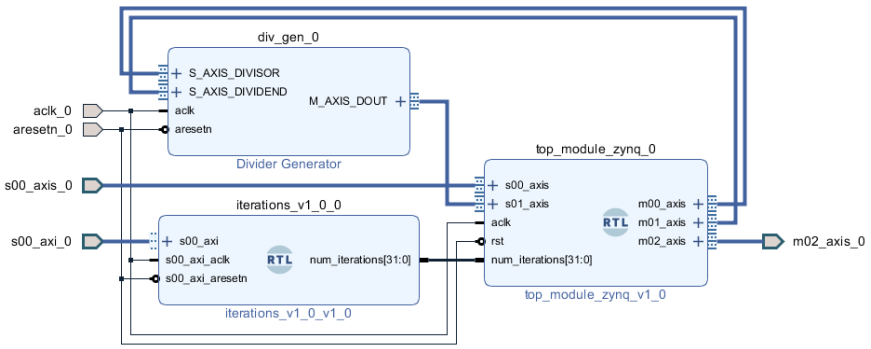


Figure B.1.1: Block diagram for Architecture-1.

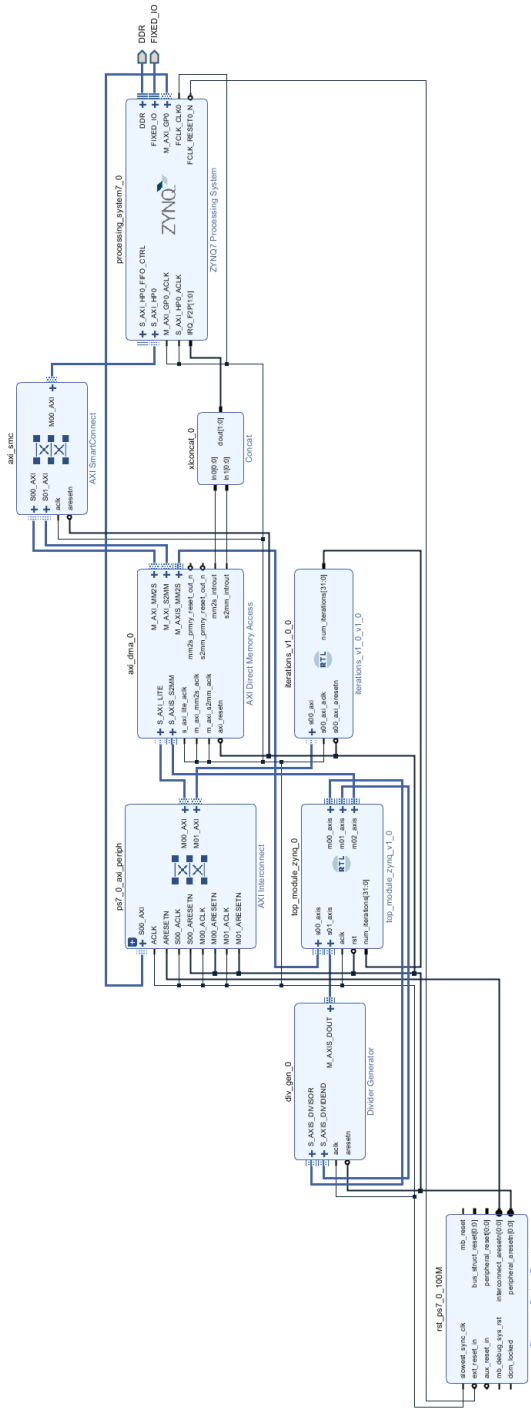


Figure B.1.2: Block diagram for Architecture-1 connected to the processing system.

B.2 Architecture-2

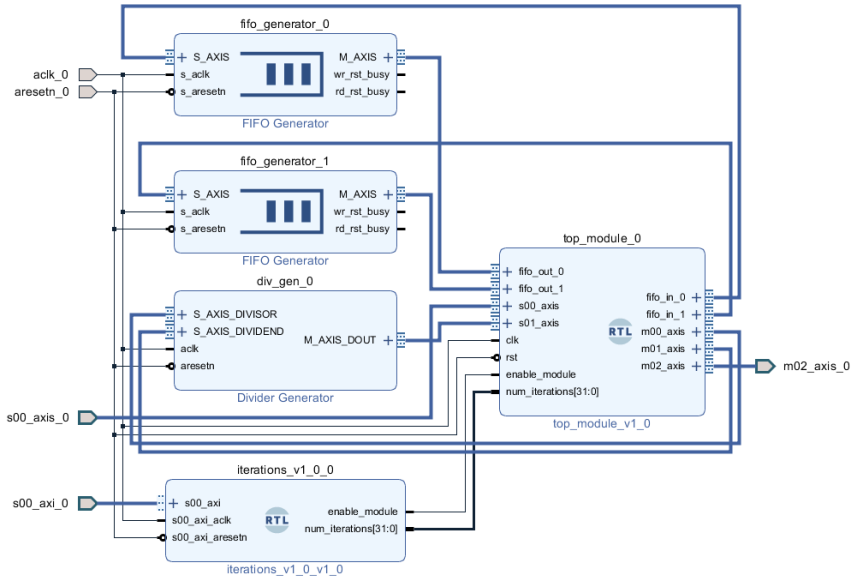


Figure B.2.1: Block diagram for Architecture-2.

Appendix C

Maximum Image Sizes

A target FPGA (e.i., Xilinx Zynq-7020) has 140 Block-RAM which results in a total of 4.9 Mib[1] of memory. In addition, when using the data width equal to o 8, 16 or 32 bits, the capacity of one block RAM is reduced to 32 Kb [49].

Architecture-1 stores three equal sized images internally in the accelerator. Two of the images has data width equal to 16-bits, and one image has data width equal to 22 bits. The maximum number of samples in one image can be found as

$$x_{max,theoretical} = \left\lceil \frac{32 \times 1024 \times 140}{22 + 16 + 16} \right\rceil = 84954 \tag{3.1}$$

Sample are stored in ab array, where each element has its own address. The address size is equal to the poser of two, therefore the real maximum number of samples is found from

$$x_{max} = \lceil \log_2(84954) \rceil = 16 \tag{3.2}$$

and consequently equal to $x_{max,real} = (2^{16} - 1) = 65535$ elements in total. For example, for an image with width equal to $M = 1200$ samples, the maximum height of the image is equal to

$$N_{max} = \left\lceil \frac{65535}{2^{16} - 1} \right\rceil = 54 \tag{3.3}$$

[1] [1] writes 4.9 Mb (megabits) and not 4.9 Mib (megabibits). There are 140 BRAMs each of size 36 Kib which in total is equal to 5160960 bits, which in terms is equal to $5160960 * 2^{-20} = 4.9Mib$.

Architecture-1 also contains 8 line buffers in the convolution block, which potentially can be implemented in BRAMs. The size of one line buffer depends on the image width. Since the number of samples in one image row is relatively low, the synthesis tool tends to implement the element in DRAMs in stead.

Architecture-2 stores only one image internally, but it does also contain two FIFOs implemented in BRAMs. The FIFO sizes depend on the image size and the maximum kernel size (i.e., equal to 9), and are found by

$$\mathrm{F0} = M \times ((W-1)/2) + 3 \qquad = M \times 4 + 3 \tag{3.4}$$

$$\mathrm{F1} = M \times (3 \times (W-1) + 8) + 52 \qquad = M \times 32 + 52. \tag{3.5}$$

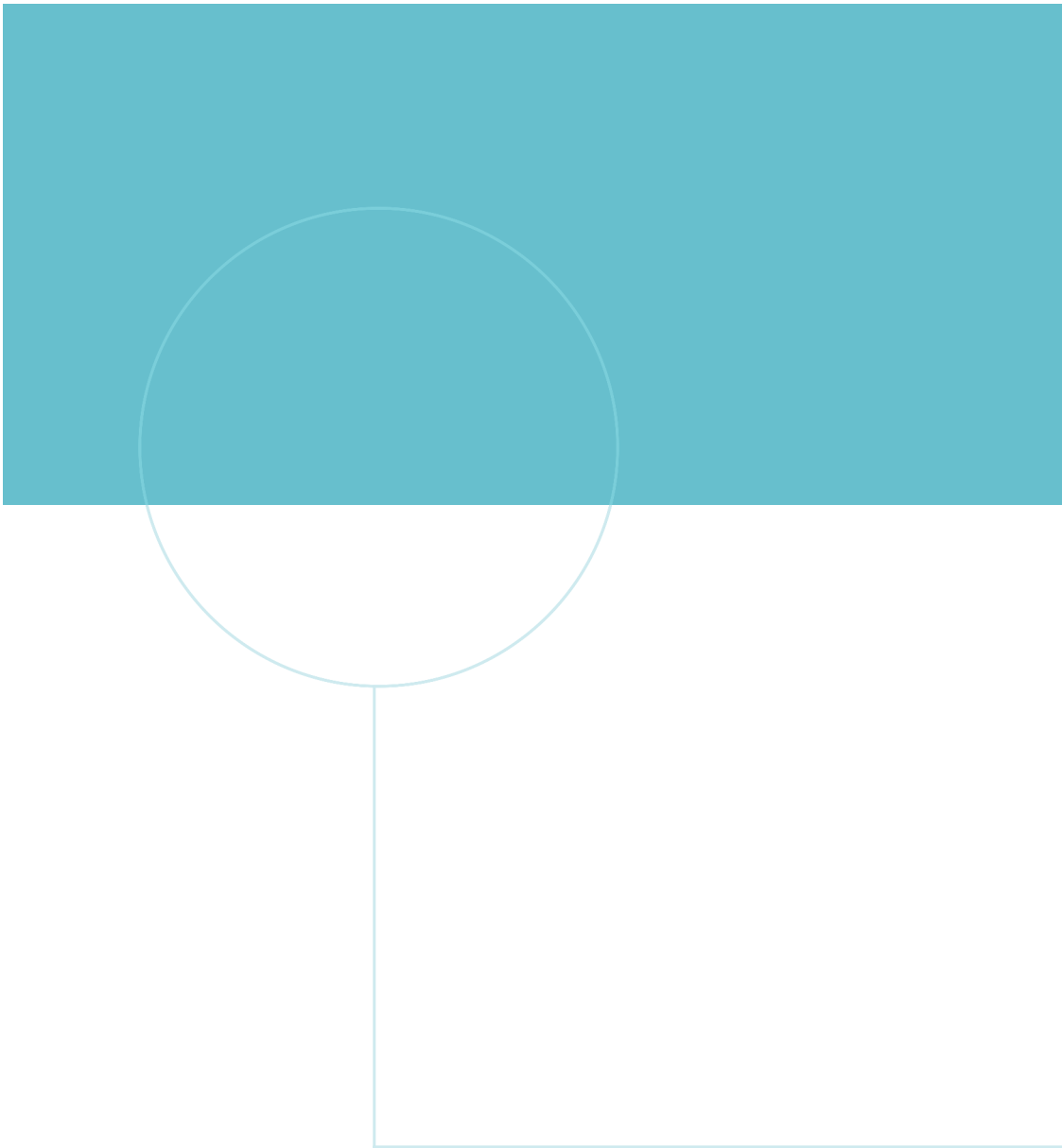
The data stored in FIFO_0 has width equal to 32 bits, where the real data is equal to 22 bits, but extended to 32 bits in order to be connected to the external Xilinx FIFO. The data stored in FIFO_1 has width equal to 16 bits and the data width of the elements stored in the BRAM is equal 16-bits. Additionally, the number of elements stored in Xilinx FIFO is a power of two. The number of samples if found from the following equations

$$\mathrm{F0}_{bram} = \left\lceil \frac{2^{\lceil \log_2(\mathrm{F0}) \rceil} \times 32}{36 * 1024} \right\rceil \tag{3.6}$$

$$\mathrm{F1}_{bram} = \left\lceil \frac{2^{\lceil \log_2(\mathrm{F1}) \rceil} \times 16}{36 * 1024} \right\rceil \tag{3.7}$$

$$M = \frac{32 \times 1024 \times (140 - \mathrm{F0}_{bram} - \mathrm{F1}_{bram}) - \mathrm{F0} \times 32 \; \mathrm{F1} \times 16}{16 \times N} \tag{3.8}$$

which for $N = 1200$ gives $M_{max} = 135$, which in turn results in a possibility to work with images containing 97200 more samples than for the Architecture-1, when using the Xilinx Zynq-7020.



NTNU

Kunnskap for en bedre verden