Master's thesis

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

Samuel Boyle

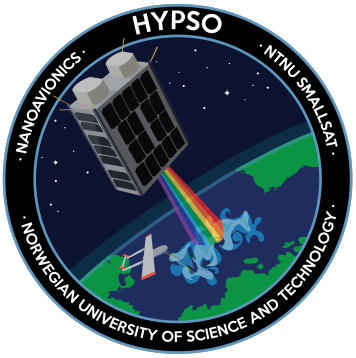
Design Space Exploration of FPGA Accelerators for Hyperspectral Anomaly Detection

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Milica Orlandić
July 2023

NTNU
Norwegian University of
Science and Technology

Samuel Boyle

Design Space Exploration of FPGA Accelerators for Hyperspectral Anomaly Detection



Master's thesis in Electronics Systems Design and Innovation
Supervisor: Milica Orlandić
July 2023

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

NTNU
Norwegian University of
Science and Technology

Abstract

Small satellites, such as those in the HYPerspectral small Satellite for ocean Observation (HYPSO) mission, represent a form of embedded system characterised by heavily constrained communication capabilities. To overcome these constraints, this thesis focuses on leveraging reconfigurable hardware accelerators to enable efficient and low-power processing of HyperSpectral Images (HSIs) onboard these satellites. The specific objective of this thesis is to address and explore hardware accelerators for state-of-the-art anomaly detection. The anomaly detection algorithm is refined into lower levels of abstraction, eventually resulting in a target implementation suitable for Field Programmable Gate Array (FPGA) acceleration using High Level Synthesis (HLS). This refinement process ensures coherency with existing literature, builds upon the high-level implementation, and provides an analysis that identifies opportunities for further improvements. Subsequently, the thesis delves into the design space of accelerator architectures, with a specific focus on the AutoEncoder (AE) component. Through a comprehensive exploration of the design space, architectures that lie on the Pareto frontier are proposed. These proposed architectures provide dominant performance in resource utilisation estimates during synthesis. A subset of the architectures are implemented and physically tested on the FPGA. The results demonstrate significant speedup, ranging from 85% to 94%, compared to non-accelerated implementations. Furthermore, the tested architectures reveal the potential for even faster designs, emphasising the need for further fine-tuning and exploration. In conclusion, by achieving substantial speedup and providing numerous potential accelerators for different throughputs this work enables efficient onboard processing of HSIs, effectively contributing to the mission objectives of the HYPSO program. These findings establish a solid foundation for the development of potential designs to accelerate the inference of anomaly detection algorithms on specific platforms, such as the ZCU-104 MultiProcessor System on Chip (MPSoC) and HYPSO-1 Zynq-7030 System on Chip (SoC). These proposed designs hold the potential to achieve a throughput of at least 140 MB/s, with resource utilisation ranging from 70% to 90% on the Zynq-7030 FPGA. Additionally providing a range of higher throughput potential for the MPSoC while simultaneously highlighting opportunities for further refinement and optimisation.

Preface

This thesis has been the final submission for the completion of my European Master in Embedded Computing Systems (EMECs). This work has been completed as part of the SmallSat lab in the Norwegian University of Science and Technology (NTNU). I am grateful for the opportunity which has been an at times challenging, but overall enjoyable experience.

I would like to thank my supervisor Milica for her support over this thesis and for our interesting discussions throughout the year. Her insightful points have encouraged me to delve deeper into the technical aspects of the Masters and develop my professional and personal skills.

I would like to thank the friends made along the way in Technische Universität Kaiserslautern and my final EMECs destination NTNU. I would also like to thank my family for their support and give a special thanks to my girlfriend for her support every day.

Acronyms and Abbreviations

ABU - Airport Beach Urban
AE - AutoEncoder
AI - Artificial Intelligence
AMBA - Advanced Microcontroller Bus Architecture
AED - Attribute and Edge preserving filters Detector
AWDBN - Adaptive Weights Deep Belief Network
BIL - Band Interleaved by Line
BIP - Band Interleaved by Pixel
BRAM - Block Random Access Memory
BSQ - Band SeQuential
CLB - Configurable Logic Block
CPU - Central Processing Unit
CRD - Collaborative Representation Detector
CRX - Causal Reed-Xiaoli
CubeSat - Cube Satellite
CU - Computation Unit
DBN - Deep Belief Network
DL - Deep Learning
DMA - Direct Memory Access
DNN - Deep Neural Network
DSP - Dedicated Signal Processor
DSW - Double Sliding Window
DWRX - Dual Window Reed-Xiaoli
FPGA - Field Programmable Gate Array
FPR - False Positive Rate
FP - False Positive
FrFE-RX - Fractional Fourier Entropy Reed-Xiaoli
GD - Gradient Descent
GRX - Global Reed-Xiaoli
GPU - Graphics Processing Unit
HDL - Hardware Description Language
HW/SW - Hardware/Software
HPC - High Performance Coherency
HSI - HyperSpectral Image
HSI-AD - HyperSpectral Anomaly Detection
HYPSO - HyperSpectral small Satellite for ocean Observation
IP - Intellectual Property
LEO - Low Earth Orbit
LUT - LookUp Table
LRX - Local Reed-Xiaoli
MAC - Multiply ACcumulate
MPAF - Morphological Profiles and Attribute Filter
MPSoC - MultiProcessor System on Chip
NN - Neural Network
NTNU - Norges Teknisk-Naturvitenskapelige Universitet
OPU - Onboard Processing Unit
PCA - Principle Component Analysis

PE - Processing Element
PL - Programmable Logic
PS - Processing System
PUT - Pixel Under Test
RE - Reconstruction Error
RBM - Restricted Boltzmann Machine
ROC - Receiver Operator Characteristics
RTL - Register Transfer Level
SDBP - Spatial Density Background Purification
SDR - Software Defined Radio
SmallSat - Small Satellite
SoC - System on Chip
SMMU - System Memory Management Unit
SRAM - Static Random Access Memory
TTM - Time To Market
TP - True Positive
TPR - True Positive Rate
UAV - Unmanned Aerial Vehicle

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 HYPSO

HYPSO, which stands for HYPerspectral small Satellite for ocean Observation, is a series of space missions conducted by the Small Satellite (SmallSat) lab at Norges Teknisk-Naturvitenskapelige Universitet (NTNU) in Trondheim. The main objective of these missions is to continuously monitor the Norwegian coast and fjords and promptly respond to the detection of ocean pollution, chemical substances, or algal blooms. By detecting these issues, the missions aim to prevent or mitigate the damage they can cause to marine life. The initial mission, HYPSO-1 was launched in January 2022 and is a SmallSat with a volume of 6 litres. It is equipped with a hyperspectral imaging payload, allowing it to capture images across thousands of channels spanning the electromagnetic spectrum. The upcoming mission, HYPSO-2, is the next focus for the SmallSat lab, aiming to launch another SmallSat with enhanced communication capabilities, particularly a software-defined radio (SDR) [1].

1.2 Motivation

Satellites offer the advantage of long-term Earth observation without the need for constant mechanical maintenance associated with unmanned aerial vehicles. Cube Satellites (CubeSats), in particular, have gained popularity due to their small size, low cost, and accessibility. Their relative simplicity and minimalist design imposes significant constraints on power consumption, weight, and volume, which in turn affect their processing power and communication capabilities. One major challenge arises when transferring HyperSpectral Images (HSIs), which are high resolution images with thousands of spectral components composing a pixel. Their large size leads to a communication bottleneck as the small satellite does not have the necessary range or bandwidth to transmit every capture for analysis. To address this issue, onboard processing of images becomes a necessity, whether to determine whether it is worthwhile to transmit to a ground station or to extract and only send essential details. To achieve this goal, the onboard processing requirements begin to involve compute-intensive algorithms and artificial intelligence (AI), which require processing power beyond a central processing unit (CPU). This would be a major limitation for achievable operations of small satellites in the HYPSO mission without dedicated processing power. Reconfigurable hardware accelerators like Field Programmable Gate Arrays (FPGAs) provide an efficient and flexible solution with lower power consumption compared to graphics processing units (GPUs).

A priority of the NTNU SmallSat lab is the creation of FPGA hardware implementations to facilitate these algorithms [2]. The particular problem of focus in this thesis is a subset of target detection, anomaly detection. Algae and pollutants may have different measurements across the electromagnetic spectrum and any single image capture provides a hyperspectral signature. Finding signatures that could indicate the presence of these is a goal of target detection. Finding signatures that vary from an expected measurement is the goal of anomaly detection. This includes different algae, pollutants or points of interest that have not yet been discovered.

1.3 Problem Definition

State of the art approaches to anomaly detection focus on combining the field of Deep Neural Networks (DNNs) with classical anomaly detection algorithms. Promising prior work by Gunderson [3] based on work by Ning et al. [4] delved into the creation of a state of the art anomaly detection algorithm which follows this AI complexity trend. Gunderson then created a partial low-level C and hardware implementation to accelerate the inference of this detector, but this was incomplete in both the breadth of possible hardware architectures for the AI component of the problem, the AutoEncoder (AE) and lacked the implementation of a spatial component, the Double Sliding Window (DSW) which has its roots from classical algorithms [5]. The existing limitations of a complete accelerated implementation pose a hindrance to achieving state-of-the-art accuracy for HSIs. It is necessary to create an implementation that can be deployed onto HYPSO satellites as part of their onboard processing requirement to maximise operational efficiency. There is a need for an implementation capable of efficiently processing HSIs. This thesis aims to develop and evaluate hardware accelerators for the goal of a complete acceleration with a variety of parameters that can be constrained or optimised depending on operational requirements.

1.4 Contributions

This thesis aims to make contributions to the high-level implementation of HSI Anomaly Detection (HSI-AD) and presents an exploration, investigation and full hardware implementation of HSI-AD using the Adaptive Weights Deep Belief Network algorithm from [3] and [4].

The key objectives are as follows

- Refine and adapt a pre-existing high-level implementation of anomaly detection algorithm; AWDBN, for coherency in literature [3], [4]. Improving the implementation's coverage to provide scoring over the entire HSI, including previously uncovered areas.
- Develop a low-level C code implementation for the anomaly detection algorithm with established model parameters used in testing functional correctness of the accelerator.
- Create new hardware architectures for a pre-existing AE implementation in High Level Synthesis (HLS).
- Create an architecture in HLS for the DSW, test it is capable of high-throughput to match the AEs.
- Perform a design space exploration of autoencoder architectures including the exploration of different achievable throughputs when integrated with a choice between two synthesised throughputs for the DSW.
- Generate a Pareto frontier in synthesis estimations for comparison of objective parameters from the design exploration.
- Implement and test accelerators on the FPGA. Evaluate results of the test against the software implementation.
- Propose possible designs for accelerating the inference of AWDBN on the ZCU-104 MultiProcessor System on Chip (MPSoC) and HYPSO-1 Zynq-7030 System on Chip (SoC) platforms.

These contributions are mainly towards facilitating the operational capacity of satellites and other edge devices with dedicated FPGA accelerators. This includes those planned and present in the HYPSO mission. Additional improvements to AWDBN, in the form of a flexible window as opposed to a single window parameter have also been suggested, which could increase the algorithm accuracy further. Based on the work carried out to achieve these objectives it is estimated that future designs would be capable of achieving a throughput of at least 140 MB/s with 70% to 90% resource utilization on the Zynq-7030 FPGA with a variety higher throughput options for the larger ZCU-104.

The findings from this thesis aim to be submitted and accepted to the 2023 IEEE Nordic Circuits and Systems Conference under the title "High Level FPGA Design of Deep Learning Hyperspectral Anomaly Detection". The results have implications for hyperspectral imaging, FPGA accelerators and anomaly detection. This submission is to facilitate knowledge transfer with other researchers in these fields.

Chapter 2

Background

2.1 Hyperspectral Imaging

Hyperspectral images taken from hyperspectral cameras collect readings at hundreds of wavelengths across the electromagnetic spectrum. These readings allow for more than just colour to be observed within the image plane, including the physical properties and materials of objects present. These images are organised familiarly as a collection pixels in a spatial 2D format where each pixel then has reflectances with a spectral signature as is illustrated in Fig. 2.1. This reflectance corresponds to the measurements of intensity along discrete points in the electromagnetic spectrum. As hundreds of readings across the spectrum for each pixel are collected, this is substantial enough to introduce a 3D component to the originally 2D image forming a "HSI-Cube". This cube is a spatial representation of the challenge it is to process these hyperspectral images which can substantiate hundreds of megabytes per capture [6]. The increased amount of data generated by these images necessitates the algorithms operating on them to be supported by dedicated processing power and the inclusion of pre-processing to eliminate complexity [1].

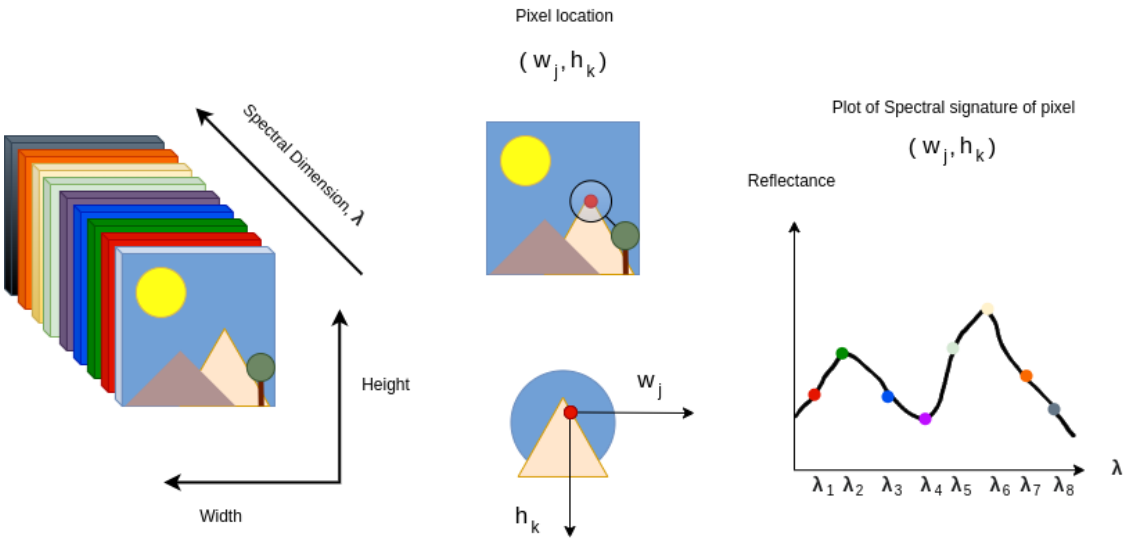


Figure 2.1: Reflectance of a point in a hyperspectral image cube

2.1.1 Remote Sensing

Remote sensing in the HYPSO project involves the continuous capture and analysis of HSIs for monitoring the Norwegian coast, observation of wildfires, deforestation or other points of interest. HYPSO also makes general captures for dataset contributions to enable researchers around the globe to study spectral properties and test algorithms [1]. To be able to meet operational requirements additional algorithms must be applied to captures to avoid unnecessary transmissions which deplete vital bandwidth. These algorithms include target detection [7], anomaly detection

[3], segmentation [8] and compression [2]. Running these algorithms on HSI cubes when accuracy and robustness is critical requires a large processing commitment. The application of these stages and algorithms make up an onboard image processing pipeline on HYPSO. The optimisation and implementation of the pipeline is currently an ongoing area of research in the HYPSO project.

2.1.2 Case Study; HYPSO-1

HYPSO-1, launched 13th of January 2023, is a small satellite. It is a 6 litre (6U) CubeSat, meaning it is made up of two columns of 3 10x10x10cm blocks stacked together. These cubes were conceptualised in the 1990s with the intention of being a standardisation and simplification to make space technology more accessible [9]. HYPSO-1 resides in sun-synchronous Low Earth Orbit (LEO). HYPSO-1 can take HSIs with resolutions above 1000x1000 at 120 bands per pixel, this is 480 MB of data for a single image. Due to its small package HYPSO only has a matter of minutes to transfer data, including HSIs after performing its telemetry and operational upkeep tasks to its base at NTNU [1]. This is an unrealistic amount of time for HYPSO to transfer all of the images it could possibly take in a day. This creates a demand for algorithms that are able to to determine which images are of interest or to extract essential information to be implemented on the satellite. These algorithms are complex and so this motivates an in-depth understanding of HSIs together with a solution involving dedicated hardware. The dedicated hardware for HYPSO-1 comes in the form of an FPGA available on its Xilinx Zynq-7030 SoC. Future HYPSO missions will feature platforms like the Ultrascale+ ZCU104 with a larger FPGA.

2.2 Using Hyperspectral Images

2.2.1 Physical Representation

Band-Interleaved by Pixel

Hyperspectral images can be stored before processing in a Band-Interleaved by Pixel (BIP) format, see Fig. 2.2. This format means that pixels and their bands are stored completely and contiguously from their lowest to highest band, a pixel at a time, from first to last sequentially. This format is advantageous for algorithms that work over a single pixel at a time, as it simplifies the memory access pattern for this case [10].

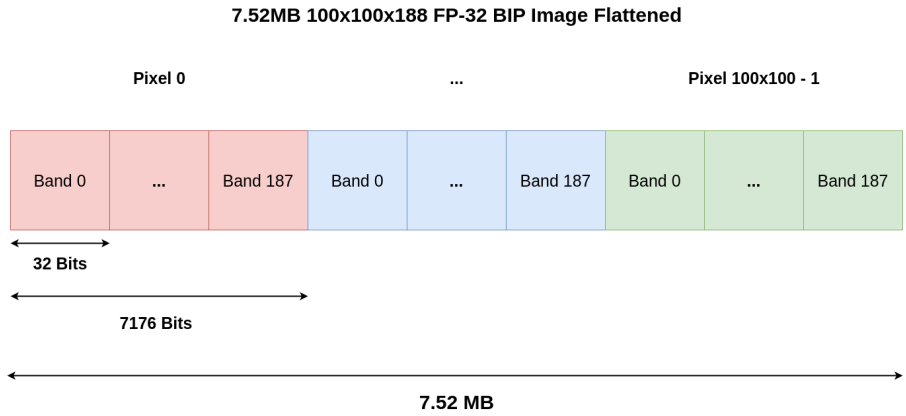


Figure 2.2: Band-interleaved by pixel format for height, width, 100x100 188 band 32 bit floating point image.

Band-Interleaved by Line

Another common representation of hyperspectral images is Band-Interleaved by Line (BIL). In this format, the bands of the pixels in the image are stored for each row contiguously as a collection of bands-following-bands. This means that the first band is stored in whole at the beginning of every row, with every row finishing with the highest band. This format provides a generalist approach of being able to access bands independently and pixels spatially by row [10].

2.2.2 Band Sequential

In Band SeQuential (BSQ) format the bands are stored contiguously for every pixel in the image as opposed to row-wise as in BIL. The image starts with the first band for every pixel and ends with the last band for every pixel. This format is optimal for accessing an entire band at once [10].

2.2.3 Reducing Complexity

Dimensionality Reduction

Due to the large size of hyperspectral images, a key pre-processing step for the management of HSIs is the reduction of the size of the reflectance dimension while preserving the information it contains. The aim is to reduce the memory footprint of the cube and its processing complexity. With Principal Component Analysis (PCA), 98% of the reflectance of a hyperspectral pixel can be distributed among 12 principal component bands [11]. This can eliminate more than 90% of the memory footprint and a majority of the processing complexity of a HSI. The target dimension can be adjusted to suit environments with varying distributions of reflectance.

2.2.4 Hyperspectral Anomaly Detection

HSI-AD, is a form of unsupervised target detection. Anomalies in HSIs are pixels that stand out by being different to what surrounds them, their background. Algorithms for HSI-AD do not know before executing what form or size an anomaly will take, only that it is an unexpected observation relative to the rest of the image. An example is displayed in Fig. 2.3, the anomalies are sparsely located at the center of the image and presumably surrounded by a grass field.

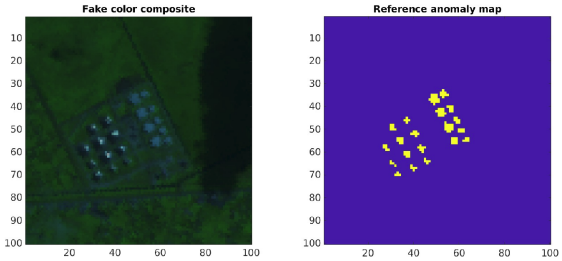


Figure 2.3: Example of fake-colour image and the anomaly ground truth from ABU dataset [12].

2.2.5 Reed-Xiaoli Baseline Detectors

Today the most well known algorithm for HSI-AD is the 1990 Reed-Xiaoli detector also known as the Global Reed-Xiaoli detector (GRX) [13]. Many HSI-AD algorithms and approaches originate from the GRX detector and it continues to be popular in research and for use as a baseline tool for HSI-AD performance bench marking and comparison [5]. This detector is an example of a statistical approach to anomaly detection. GRX applies a multivariate Gaussian distribution over the entire image by generating means and covariance estimates over image subsections to create a background distribution. Anomalies are detected by Mahalanobis distance from this background distribution [14]. With probability distribution Q, with mean μ, and a inverse covariance matrix C^{-1},

$$D_{GRX}(x, Q) = \sqrt{(x-\mu)^T C^{-1}(x-\mu)} \tag{2.1}$$

When score D passes a determined threshold, the pixel x can be classified as an anomaly.

A proposed improvement to GRX; Local Reed-Xiaoli (LRX), considers that for local estimates of anomaly scores for pixels it is best to center the Guassian distribution region around each pixel individually rather than fewer pre-defined subregions [5].

$$D_{LRX}(x, Q_{win}) = \sqrt{(x-\mu_{win})^T C_{win}^{-1}(x-\mu_{win})} \tag{2.2}$$

This is where the mean, covariance, distribution are taken from a window region around a center Pixel Under Test (PUT). This detector has unclear origins but is mentioned in an array of literature

[5], [15]. This is implemented by a sliding window operation which is shown in Fig. 2.4. This means for every pixel the window will slide down through the columns, at the end of each row it will begin from the first column but shifting the the rows upwards until all the pixels are scored.

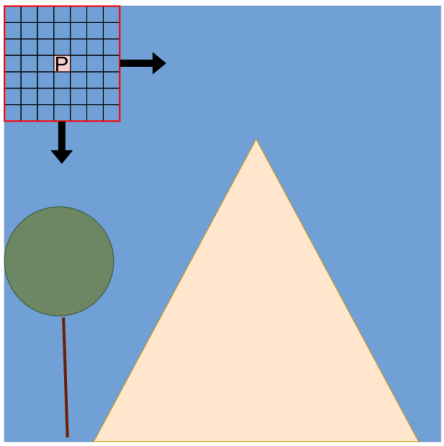


Figure 2.4: Local Reed-Xiaoli sliding window

To prevent local anomalies in the image contaminating the pixel distance calculation the DSW was introduced, meaning that there is an interior sub-window inside the sliding window, see Fig. 2.5. Any pixels caught in this interior window will not contribute to the Guassian distribution or Mahanalobis distance calculations.

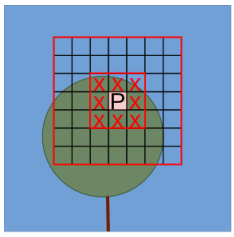


Figure 2.5: Local Reed-Xiaoli double sliding window

Following the DSW strategy a Dual Window RX (DWRX) uses a DSW where the inner window is used as a mean for the pixel under test [16]. These LRX and DWRX variations are particularly prohibitive in their ability to have real-time implementations as there is an image subsection processed for every pixel **rx**, [15].

2.2.6 Aiport-Beach-Urban Dataset

The Airport-Beach-Urban (ABU) is a dataset from Xudong Kang [12] and it is a popular dataset for performing benchmarking and comparison of HSI-AD algorithms. It features hyperspectral images in airport, beach and urban environments which feature different shapes and sizes of anomalies in a realistic imaging scenario. Table. 2.1 shows the image dimensions, alongside their number of bands. Fig. 2.6 shows an example of ABU image air 3 with its ground-truth for anomalous pixels.

Image	Height/Width	Bands
Air 1	100x100	205
Air 2	100x100	205
Air 3	100x100	205
Air 4	100x100	191

Image	Height/Width	Bands
Beach 1	150x150	188
Beach 2	100x100	193
Beach 3	100x100	188
Beach 4	150x150	102

Image	Height/Width	Bands
Urban 1	100x100	204
Urban 2	100x100	207
Urban 3	100x100	191
Urban 4	100x100	205
Urban 5	100x100	205

Table 2.1: ABU Airport, Beach, Urban characteristics

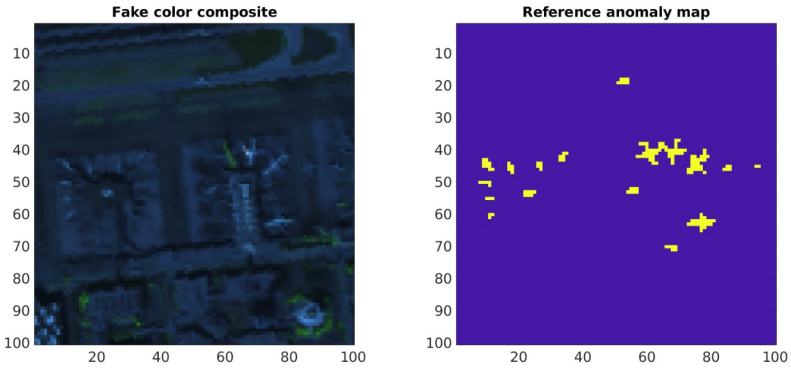


Figure 2.6: ABU dataset image air 3 with ground truth

2.2.7 Detector Evaluation

The standard performance metric for anomaly detection is Area Under the Receiver Operator Characteristics (ROC) curve, abbreviated to AUC. AUC provides an estimation of the probability of a random positive being classified before a random negative. The AUC reflects the accuracy of the anomaly detection method used by comparing what is classified as an anomaly when using a ground-truth map compared to a given anomaly-score map. In AUC this is done without committing to a decision threshold as it averages over all thresholds. Different thresholds for anomaly classification can lead to different false positives, so this means averaging the true positive rate over all possible false positive rates [17]. A True Positive Rate (TPR) is the ratio of True Positives (TP) to all positives by using detected False Negatives (FN), for a given threshold.

$$F_0(threshold) = \frac{TP}{TP + FN} \tag{2.3}$$

The converse is the False Positive Rate (FPR) where this is the rate of False Positives (FP) classified for all negatives by using detected True Negatives (TN) for the threshold

$$F_1(threshold) = \frac{FP}{FP + TN} \tag{2.4}$$

These TPR and FPR are calculated for a large number of classification thresholds, generating an ROC curve. The shape of this curve has implications on performance, with its vertical axis as the TPR rate and horizontal the FPR; the area under the curve becomes the evaluation metric.

$$AUC = \int_0^1 F_0(s) dF_1(s) \tag{2.5}$$

This would be applied to a graph as pictured in Fig. 2.7.

Overview of state of the art anomaly detectors

Before introducing the chosen algorithm for anomaly detection, it is useful to know different detectors from literature to help interpret Table 2.2. These detectors are summarised below.

- Causal RX (CRX): CRX aims to achieve real-time processing by using a sample correlation matrix to compute the anomaly score for each pixel [18].
- Kernel RX (KRX): KRX is a non-linear implementation of the RX algorithm that maps pixel vectors to a higher-dimensional feature space using a kernel function, such as the radial basis function kernel [19].
- Fractional Fourier Entropy RX (FrFE-RX): FrFE-RX applies the fractional Fourier transform to extract features before applying the RX algorithm. This approach aims to reduce noise and create a greater distinction between anomalous and background pixels [20].

- Collaborative Representation Detector (CRD): CRD assumes that background pixels can be represented by surrounding pixels spatially, while anomalous pixels cannot. It computes a weight matrix that minimises the difference between a pixel and its surrounding pixels, and the anomaly score is calculated based on this representation [21].
- Spatial Density Background Purification (SDBP) Detector: SDBP combines a density peak clustering algorithm with the CRD algorithm to reduce anomaly contamination [22].
- Attribute and Edge-preserving filters Detector (AED): AED applies attribute filters to separate anomalies from the background and uses an edge-preserving filter for post-processing to improve detection accuracy [23].
- Morphological Profile and Attribute Filters detector (MPAF): MPAF reduces dimensionality in the spectral domain using morphological profiles and applies attribute filters to identify and filter anomalies [24].
- Deep Belief Network (DBN) algorithms utilise AEs, which are neural networks trained to compress and decompress input data. By reconstructing hyperspectral pixels, they can detect anomalies based on higher reconstruction errors for anomalous pixels compared to background pixels. This includes DBN, AWDBN and Novel-AWDBN which will be discussed more in a later section [4], [3].

2.2.8 Detector Speed and Accuracy

The two most important metrics for detectors is the accuracy and speed at which they can process HSIs. Since most detection takes place in the domain of remote sensing, this means it will be performed on satellites or Unmanned Aerial Vehicles (UAVs). As an embedded system constrained in terms of processing power and hardware availability, these algorithms need to be fast, lightweight or have dedicated hardware [1]. Table 2.2 provides an overview of HSI-AD processing time including some classical and some state of the art algorithms.

Algorithm	MPAF	AED	FrFE-RX	LRX	GRX	CRD	SDBP	DBN	AWDBN	Novel-AWDBN
ABU AUC (Avg)	0.9916	0.9757	0.9657	0.9604	0.9420	0.9673	0.9872	0.9637	0.9721	0.9870
Runtime (s)	0.17	0.41	22.89	57.59	0.14	39.25	7637.45	3.64	3.86	3.89
Source of measurement	[24]	[23]	[20]	[23]	[23]	[23]	[22]	[3]	[3]	[3]

Table 2.2: Comparison of detector accuracy and computation times.

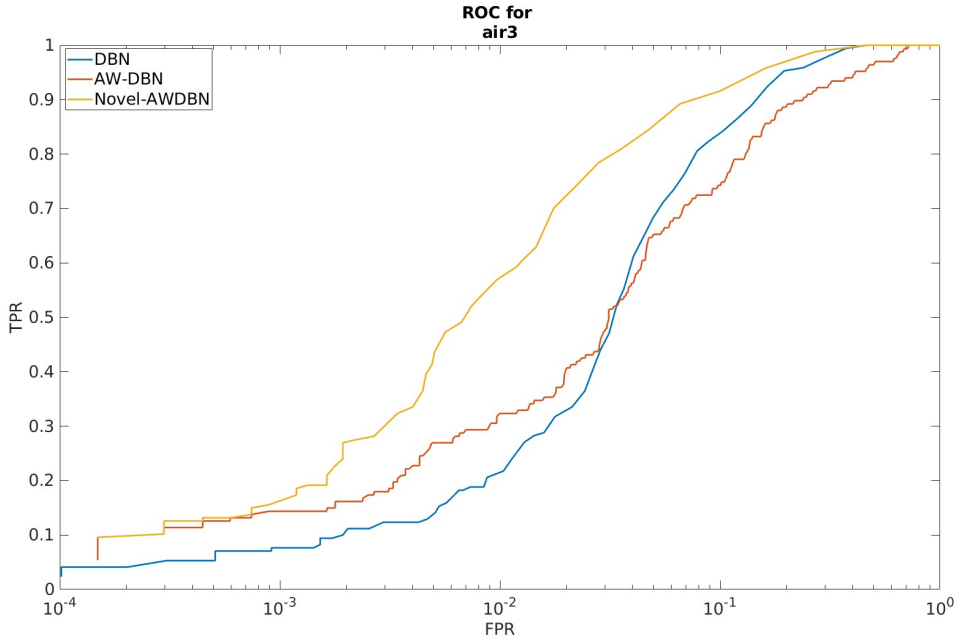


Figure 2.7: ROC curve from algorithms implemented in prior work by Gunderson [3]

2.2.9 Choice of Algorithm

This thesis focuses on a design space exploration for dedicated hardware acceleration for AWDBN based anomaly detection. This algorithm features a spatial dual window as mentioned in LRX combined with an AI approach rather than a Guassian based RX. AWDBN and the Novel AWDBN have shown promising performance and are state of the art in the beach 1, beach 3 and urban 1 scenes in particular, with a measure AUC of 0.9999, 0.9995 and 0.9989 respectively as highlighted in prior work by Gunderson [3]. The spatial nature of the DSW and the Deep Learning (DL) nature of this algorithm means hardware acceleration is required for handling large images. AI and sliding windows map well to hardware acceleration in particular as will be presented in the continuation of this background material and more in-depth in the literature review for FPGA accelerators.

2.3 Artificial Intelligence and Deep Learning

With the rise of DL, it has become an important component in recent solutions to HSI-AD, including the prior work which this thesis develops upon [3]. Neural Networks (NN) are inspired by the human brain and attempt to model it with interconnected artificial neurons organised into layers or varying architectures. Each neuron applies a mathematical transformation onto its input synapses. It is in the training of these networks that they become favourable for pattern recognition and prediction [25]. DL is a sub-field of NNs that aim to use multiple stacked layers combined to perform more complex tasks, see Fig. 2.8 for the DL subset of AI.

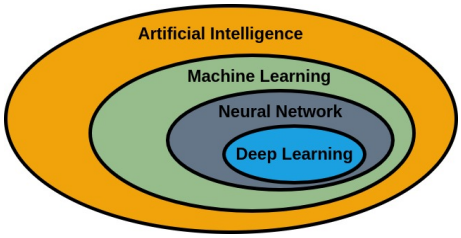


Figure 2.8: AI heirarchy

2.3.1 Deep Belief Networks

DBNs are a deep learning architecture in which the network is composed of Restricted Boltzmann Machines (RBMs). RBMs take the form of an undirected bipartite graph with one layer of nodes connected to another layer with no connections between nodes within the same layer. Fig. 2.9 provides a visualisation of a RBM, and of a DBN as a collection of stacked RBMs. In this type of architecture each of the inputs are individually mapped to each of the neurons. This means that there is no differentiation spatially in terms of the relation of input nodes, all input edges are treated as equally distanced. What makes the DBN is the stacking of these RBMs, such that the output of one layer is the input of another. This is especially useful for serving as an AE if there is a variation in input and output sizes between layers [25].

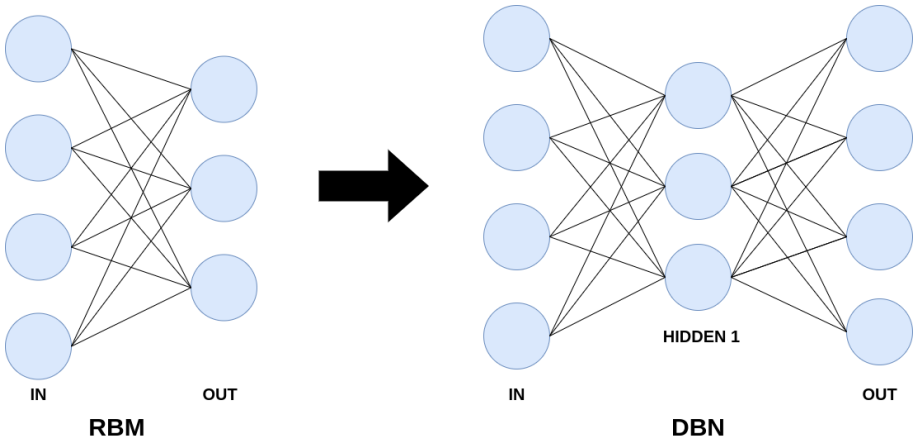


Figure 2.9: Deep belief network as composition of restricted Boltzmann machines

2.3.2 Layers and Organisation

Fully Connected

A fully-connected (FC) layer, see Fig. 2.10, is a type of RBM in which every node in the input layer has a weighted edge to every neuron in the output layer. This neuron will take for every input node a weighted edge product and accumulation. This is known as a multiply-accumulate (MAC) operation which is performed before applying a final bias offset. A non-linear activation can then be applied to the output [26]. These weights and biases are then programmable such that the network can be adjusted over a period of inferences to provide a desired functionality for a chosen input. The equation of the FC layer has *Co* as the set of output nodes and *Ci* as the set of input nodes. *W* represents the weights for each input to an output neuron at index [*Cout, Cin*] and *b* is the output node bias.

$$\begin{gathered}\mathbf{O}\left[c_o\right] = \sum_{c_i=0}^{C_i-1} \mathbf{W}\left[c_o, c_i\right] \mathbf{I}\left[c_i\right] + \mathbf{b}\left[c_o\right] \\ 0 \leq c_o < C_o, \quad 0 \leq c_i < C_i\end{gathered} \tag{2.6}$$

This equation is from [26]. As FC layers require every input is connected with every output neuron, this is disadvantageous for large inputs such as high-resolution images. Additionally as there is no inherent spatial consideration when every input is connected to every output.

Normalisation

Before an input is placed into a NN it is typically pre-processed for normalisation. This means it is transformed to have a mean of zero and unit variance. This is useful for the model as all its inputs are within the same range. In NNs incorporating a Sigmoid or Softmax function which are saturating it is beneficial to reduce early saturation this way [26]. Normalisation is not a necessity and NNs can operate well without it.

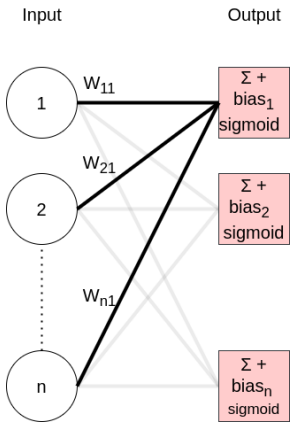


Figure 2.10: Fully connected layer with Sigmoid activation function [26].

Non-Linear Activation

The activation function used in prior work is Sigmoid [3]. The Sigmoid function takes an input value x and produces an output value y between 0 and 1, representing the probability associated with that input.

$$y = \frac{1}{1 + e^{-x}} \tag{2.7}$$

This is especially useful in NNs where probability is a concern, and thus is the function best suited for anomaly detection.

2.3.3 Autoencoders

AEs are a restricted form of DBN that can be trained to introduce a reconstructive property. It features FC layers that are arranged such that the number of input nodes are greater than the middle hidden layer. This input is exactly equal in size to the output. The key to achieving a reconstruction is to train the network to minimise the difference between its output and the input. This results in a compression of the input into a hidden code layer, known as the code. This code is then used in an FC layer followed by a Sigmoid activation function to regenerate the output. This acts as a form of lossy-compression which prioritises the accurate reconstruction of data that it is more regularly exposed to, see example in figure 2.11 . In the case of anomaly detection this means it becomes better at reconstructing pixels less likely to be anomalies. This can allow trained AEs to become a substitute for statistical methods [25].

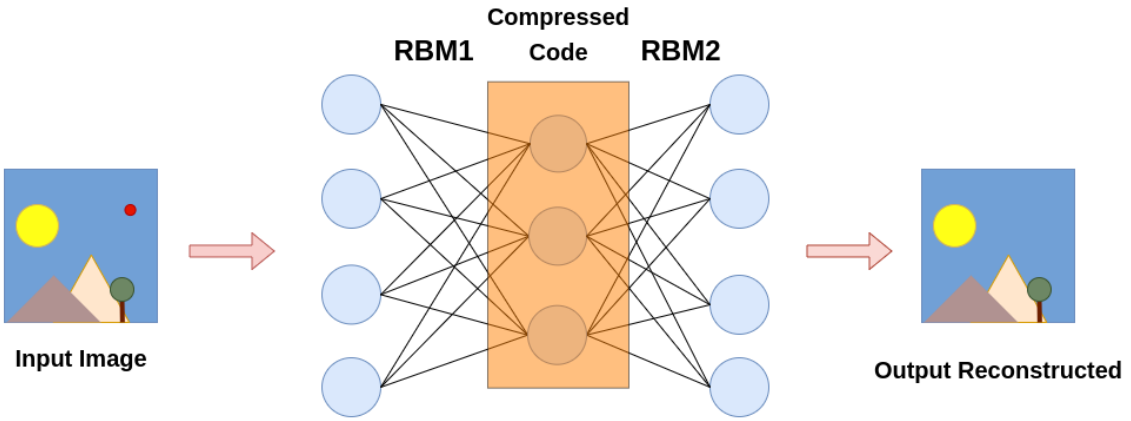


Figure 2.11: Autoencoder reconstructing image excluding poorly represented red dot.

2.3.4 Training

Training refers to the generation of necessary parameters for a NN to achieve its desired functionality. In the context of an AE these parameters constitute the weights used for MAC operations and the post-sum bias. Training generally makes alterations to the weights and biases over many iterations of the network to converge towards an accuracy limit. This accuracy is measured by a loss function for each iteration.

Gradient Descent and Back Propogation

Backpropagation is a popular choice to compute the loss function in a feed-forward NN. Backpropagation moves backwards through the NN computing derivatives which form the loss function. Training is then achieved through Gradient Descent (GD) which uses a gradient of the loss function to move parameters in a direction that descends upon this limit [27].

Contrastive Divergence

More advanced training for AEs features pre-training using Contrastive Divergence (CD). With CD each RBM is individually trained to learn some patterns of its input before being trained with GD. This can improve the performance of AEs but requires more time to be trained. The focus in this thesis is creating a strong basis for acceleration of the inference as opposed to the training. The implementations of CD and GD are prior work by Gunderson for the purpose of DBN, AWDBN and Novel-AWDBN based anomaly detectors. More information on the training algorithms are available in Gunderson's thesis [3].

2.4 State of the Art in Anomaly Detection

This thesis is a continuation of the work carried out by Aksel Gunderson [3] which was founded by work from Ma et al. [4], [28], [29] . This work introduced methods for anomaly detection using aforementioned AEs. Ma et al. introduced a method of using AEs for generating a reconstruction error [4]. Continued work from Ma et al. then proposed the combination of this with a DSW and the utilisation of the code-layer as a spectral distance with a combined reconstruction error (RE) weighting strategy [28]. Building upon this Gunderson's thesis introduced a novel weighting strategy for the anomaly score calculation of each pixel with the implementation of an improved training strategy featuring CD [3]. Work up to this point has shown state of the art performance for anomaly classification in some scenes of the ABU dataset and an average performance above many well performing anomaly detection algorithms.

2.4.1 Autoencoder Anomaly Detection

The motivation of using AEs for anomaly detection comes from the AE becoming statistically better at reconstructing pixels it has been exposed to during training. By the logic that anomalies are a rare case unlike their surrounding background, they will be poorly represented and thus not reconstruct well [4]. Encoding and decoding is performed on a pixel-by-pixel basis. Once the image pixels have been encoded and reconstructed, the reconstructed image is compared to the original image. Common pixels should be similar to the input but less represented reflectances will be reconstructed poorly and have a large deviation from the input. The RE of a pixel is the bandwise euclidean distance between the original pixel and its reconstructed pixel. RE forms the anomaly score directly in DBN-AD. For pixel x, the Eq. 2.8 encompasses scoring the pixel with input i, r reconstruction for B bands.

$$D(x) = RE(x) = \sqrt{\sum_{b=0}^{b<B} (i_b - r_b)^2} \tag{2.8}$$

2.4.2 Spatial Autoencoder Anomaly Detection

Ma et al. [28] introduced the idea of pairing an AE with the classic DSW. This pairing enables the AE to become a substitute for statistical methods including the Gaussian distribution of LRX [4].

After the AE is trained, this reconstruction based approach is able to be applied to every pixel of the image. The DSW, see Fig. 2.12 is composed of an inner-window of ignored pixels and an outer window. The pixels inbetween the inner and outer windows are known as neighbourhood pixels. Neighbourhood pixels contribute to anomaly score calculations through their reconstruction error and code layer representations, which are used with the RE and code representation of the PUT to allocate an anomaly score.

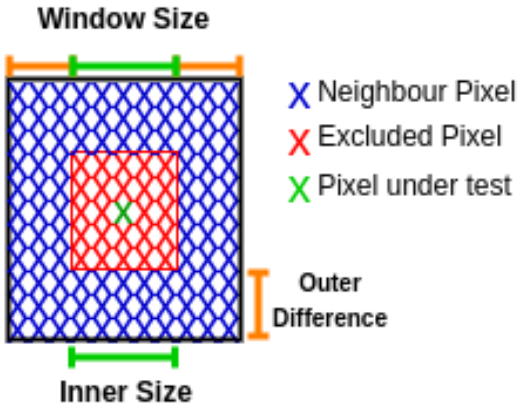


Figure 2.12: Double sliding window with PUT.

The first step is the training the AE which can include CD and GD. Once trained the inference of the AE is as described in Fig. 2.11, with FC layers, Sigmoid activation functions and the generated reconstruction errors as mentioned prior in Eq. 2.8. With a corresponding map of REs for each pixel, the code layer representation for every pixel when it was decoded also forms a code map, for which the dimensions and indexes correspond to the original image. It at this point the sliding window is ready to be applied over every pixel in the image. The sliding window involves five stages of processing.

1. RE mean calculation
   Firstly, from the set of Neighbourhood REs, they are averaged, their average is the summation and division by the number of neighbours. This is the average neighbourhood reconstruction error.

$$RE_\mu = \sum_{n=0}^{n<|N|} \frac{RE_n}{|N|} \tag{2.9}$$

2. RE standard deviation
   Once the mean is calculated, the standard deviation is calculated by root of the average squared difference from the mean for every neighbourhood pixel.

$$RE_\sigma = \sqrt{\frac{\sum_{n=0}^{n<|N|}(RE_n - RE_\mu)^2}{|N|}} \tag{2.10}$$

3. Weight matrix
   If a neighbourhood RE varies from the mean by more than a standard deviation it has its weight contribution penalised by a predetermined factor P. This factor is intended to prevent or mitigate anomalies in the neighbourhood from contributing to the anomaly score calculation. The weight matrix for all neighbours contains the inverse of their RE multiplied by this penalty factor.

$$\forall n; n \in Neighbours, \quad W_n = \begin{cases} \frac{P}{RE_n}, & \text{if } RE_\sigma > |RE_n - RE_\mu| \\ \frac{1}{RE_n}, & \text{otherwise} \end{cases} \tag{2.11}$$

4. Code distance calculation
   The code distance from the PUT is calculated for every neighbourhood pixel. This is the root of squared differences of every code band, the euclidean distance. These code distances are used for generating the anomaly score. C is the number of code bands, c is the band index, PUT corresponds to the central pixel of the window and this is calculated for each neighbourhood pixel, n.

$$\forall n; n \in Neighbours, \quad D_n = \sqrt{\sum_{c=0}^{c<C} (PUT_c - n_c)^2} \tag{2.12}$$

5. Anomaly score
   The final step is the anomaly scoring for the pixel. The code distance is multiplied by the corresponding weight for each neighbour and these are summed. After summation the score is divided by the number of neighbour pixels to create the final score.

$$D(x) = \sum_{n=0}^{n<|N|} \frac{D_n W_n}{|N|} \tag{2.13}$$

2.4.3 Novel Weight Strategy

In Gunderson's thesis [3] a new strategy for anomaly scoring was introduced where neighbour weights are not their reciprocal RE with the opportunity for penalties alone. It was modified to be the reconstruction error of the PUT multiplied by the inverse neighbour RE with the opportunity for penalties as can be seen in Eq. 2.14. This was shown to perform favourably compared to the original implementation in 2.2, with the average AUC improved from 0.9721 to 0.9870.

$$\forall n; n \in Neighbours, \quad W_n = \begin{cases} \frac{RE_{PUT}P}{RE_n}, & \text{if } RE_\sigma > |RE_n - RE_\mu| \\ \frac{RE_{PUT}}{RE_n}, & \text{otherwise} \end{cases} \tag{2.14}$$

2.4.4 Computational Cost

Novel-AWDBN can take 3.9 seconds to execute for an ABU dataset image as established in 2.2. HYPSO images are larger than this 100x100 pixel dataset, from 100x100 to 1000x1000. Being thousands of pixels per capture, this immediately requires a 100x throughput just to match the 4 second detection time. The use of this algorithm for HSI-AD in a constrained embedded satellite is difficult in terms of the CPU time commitment and power usage. This algorithm involves complex summations and multiplications which, when paired with having to fetch instructions from memory, impacts negatively on overall system power consumption. The possibility of a solution that does not require a fixed-point data representation will be important for maximising throughput while minimising power consumption. Thus from this point onwards the focus of the thesis will be on the resources available within the satellite's onboard heterogeneous SoC, the Onboard Processing Unit (OPU) to alleviate these issues [1].

2.5 Embedded Heterogeneous Computing

The HYPSO project is focused on leveraging small satellites equipped with integrated sensors, subsystems and SoCs. Once launched it is not possible to add new physical hardware, emphasizing the importance of carefully planning the equipment and system specifications required to accomplish the goals of the HYPSO project. This falls within the realm of embedded computing systems. With the successful launch of HYPSO, it is crucial to comprehend the rationale behind the decisions made regarding its onboard processing capabilities for handling HSIs and explore how this knowledge can be applied to HSI-AD [1].

2.5.1 Dedicated Accelerators

Hardware accelerators are necessary when there is a processing throughput requirement paired with a time constraint that cannot be satisfied by a CPU. This includes applications within image processing, graphics, encryption and compression. Hardware accelerators can come in different forms, typically highly adaptive, fast but power-prohibitive GPUs, high performance, low power but non-configurable ASICs and reconfigurable FPGAs. Which of these choices is preferable depends on multiple factors including the dynamics or changing nature of problems encountered, the availability of power and the required throughput. Fig. 2.13 displays properties of different hardware accelerators and their suitability for various general problem solving. There is also an Time-To-Market (TTM) and pricing element associated with these devices.

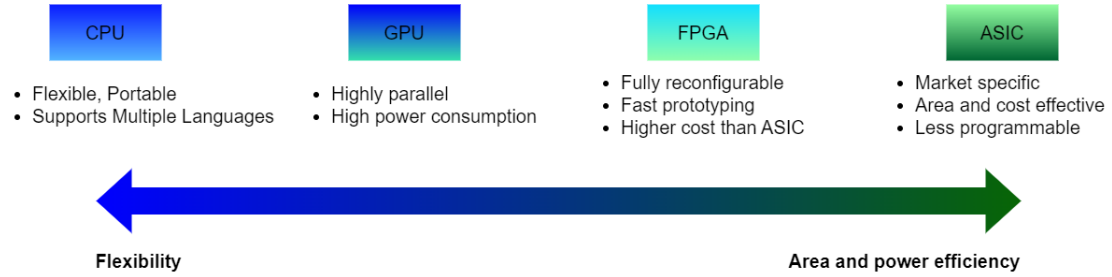


Figure 2.13: Accelerator solution tradeoffs adapted from [26].

Acceleration Solution

The solution for HYPSO and its constellation are reconfigurable FPGAs. The reasons for this choice are outlined below, which address the pros and cons of the various main acceleration solutions.

- GPUs provide flexibility but are prohibitive in terms of their power consumption. HYPSO has to generate solar power while in orbit and has no fixed supply.
- ASICs are not adaptable enough to suit an environment where they cannot be exchanged when there are a plethora of different algorithms to be run for different tasks associated with maximising SmallSat operations.
- FPGAs provide the necessary reconfigurability to handle the range of algorithms that HYPSO needs to process onboard. These algorithms can be chained in a processing pipeline, either by having multiple designs insantiated in the FPGA or through full or partial reconfiguration of the system. This provides an opportunity for exploration of the different configurations that the HYPSO OPU can occupy.

Accommodating an FPGA into an SoC is not simple. It requires knowledge in many disciplines across digital design, software and system engineering. Important concepts will now be covered regarding the design and making the most of a dedicated hardware accelerator.

2.5.2 Hardware-Software Codesign

Hardware-Software (HW/SW) Codesign is an approach to computing system organisation that intends to exploit and leverage dedicated hardware performance with software intelligence and flexibility to meet system-level objectives through their collaborative and concurrent design. This provides many advantages, firstly this allows for a faster execution of intense processing tasks by offloading these to an accelerator. Secondly this accelerator being a dedicated circuit or device will have an advantage in power required over a CPU. This allows the fulfilment of soft deadlines that maximise its functionality. When it comes to evaluating the candidate tasks and mediums to accelerate these, it is important to consider power, speed, system or area resources and scalability. Design exploration is where all of these factors are considered and explored in the hopes of generating a satisfactory design [30].

2.5.3 Design Space Exploration

Design space exploration is the process of exploring and assessing different design options and configurations to generate the most suitable implementation for a specific task or set of tasks. This involves evaluating a wide range of design parameters including hardware/software architecture, resources, memory organisation, power consumption, performance and cost. In the pursuit of an optimal system-level architecture there is also a refinement of the software and hardware implementations, which can be done with algorithms or with expert knowledge [31]. When trying to find an optimal solution between two conflicting objectives, such as power and performance or area and execution time, the Pareto principle is a useful concept. The Pareto principle states that it is possible to improve one objective without compromising the other, but it is not possible to simultaneously optimise both objectives. In the context of a Pareto graph, solutions are represented as points, and the graph shows the trade-off between the two objectives. The Pareto frontier, illustrated in Fig. 2.14, represents the set of solutions where it is not possible to improve one objective without sacrificing the other. Any solution located on the Pareto frontier is considered optimal, as it cannot be dominated by another known solution in terms of both objectives simultaneously.

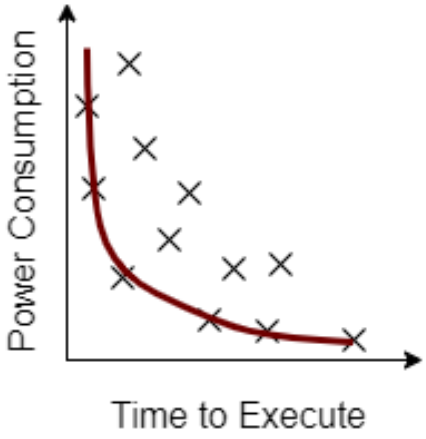


Figure 2.14: Adaptation of Pareto graph from [31].

2.6 Field-Programmable Gate Arrays

FPGAs emerged in the 1980s. The term "Field Programmable" is because of the Static Random Access Memory (SRAM) they contain which enables their functionality to be reprogrammed [32]. They are made up of a collection of different logic blocks that have to be routed through the FPGA. The arrangement and properties of these blocks are vendor specific, the focus of this thesis is on Xilinx devices. Making efficient use of these blocks while avoiding routing issues or unnecessary resource utilisation is important for creating FPGA hardware accelerators. These resources are typically organised in an "Island Style" architecture, forming a sea of configurable resources. These resources exhibit a degree of heterogeneity, providing a combination of general-purpose and specialised functionalities to cater to diverse application needs.

2.6.1 Key Concepts

Logic Blocks

Xilinx FPGAs are equipped with Configurable Logic Blocks (CLBs), which consist of Lookup Tables (LUTs), Flip Flops (FFs), and multiplexers collected into a slice. Each slice typically contains multiple CLBs. These CLBs offer programmable memory functionality capable of performing computations involving up to N inputs. An N-input LUT serves as a programmable memory element capable of computing various functions of up to N inputs. Additionaly, the D-FFs within the CLBs can store the output of these LUTs or buffer data during positive clock edges [33].

On-Chip Memory

On-chip memory is distributed through various mediums of the FPGA which have varying densities and throughputs. Block-RAMs (BRAMs) are a dense form of memory within the FPGA that are designed for large amounts of data accessed at lower bandwidths. Registers within the FPGA are implemented through fast but less dense FFs. Both can be partitioned at the cost of memory resources to increase the read/write ports for higher bandwidth applications [33].

Specialist Processing slices

Of the specialised slices in Xilinx FPGAs, the most important for processing applications is the Digital Signal Processing 48 (DSP48) slice. This offers dedicated multiply-add functionality. DSP48 slices are designed to provide efficient multiply and addition units, combined with registers for buffering and storage, making them particularly well-suited for digital signal processing tasks [34]. From Fig. 2.15 A and B are the operands used in the dedicated hardware multiplication, and C is used for a post-multiply accumulation. The input D varies depending on the specific configuration of the DSP48E2 slice but can be used for addition or shifting. This provides flexibility which allows it to be tailored to a wide range of applications.

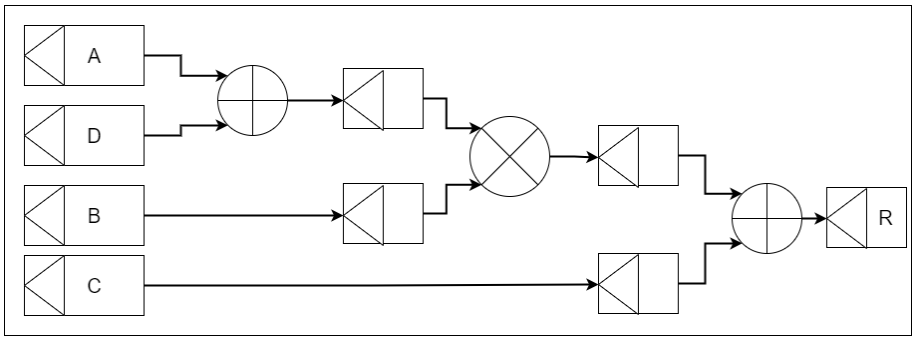


Figure 2.15: DSP48E2 slice [34].

Routing

Routing is accomplished through a network of programmable interconnects that allow signals to flow between different components within the FPGA. The routing architecture in FPGAs is typically based on a mesh or grid-like structure, known as a programmable interconnect matrix. This matrix consists of horizontal and vertical interconnect lines that form a grid pattern across the FPGA fabric. Logic blocks and specialised resources are positioned at the intersection points of these interconnect lines similar to islands. Ensuring a design is routable through the FPGA depends on many constraints over the development and tool implementation, including utilisation, timing constraints and resource placement [33].

Synthesis and Implementation

Once a high-level hardware description is written, it must be converted into a gate-level representation to be implemented in the FPGA, which is known as synthesis. Operations are performed to improve the performance, resource utilisation and power consumption during this phase. These optimizations include logic minimization, technology mapping, constant propagation, and resource sharing. Implementation is the stage where the synthesised design is mapped onto the target FPGA device. The most important part of this process is the place and route; placement of resources around the FPGA which then must be routed. Even if a design is synthesisable it may not be routable [33].

2.7 FPGA Development Flow

Both HLS and HDLs serve the purpose of translating a high-level algorithm or behaviour into a refined and optimised representation for hardware implementation. Regardless of the specific

technique chosen, the objective remains the same: to convert an abstract, high-level description of the desired functionality into a hardware model that accurately reflects the intended behavior and produces consistent results. The first step, regardless of the technique, is the refinement of the chosen algorithm into a reference model. This model serves as a validated representation of the intended results of the hardware implementation for subsequent verification and implementation stages [35].

2.7.1 Traditional Development

The traditional method of FPGA development uses Hardware Description Languages (HDLs) to create processing architectures. This method has the flexibility to allow for highly-optimised designs to be created through the experience and expertise of its designers. However, there are also drawbacks to using this approach:

1. Register Transfer Level (RTL) HDL designs often take a long time to make and are difficult to adapt to changing requirements.
2. There is little opportunity for exploration and changes once the initial architecture is proposed, any solution may be not yet be at its local or global optimum when a change could further optimise it.
3. Functional testing and verification composes a majority of the project time when RTL test benches are used.

The ideal design has to be understood and implemented from the beginning and any debugging is done through low-level simulations. If the implemented design has errors or does not perform as expected there is a risk the design will have to be re-implemented and any Computation Units (CUs) will need to be recreated. This presents a rigid approach to hardware development.

2.7.2 Development with High Level Synthesis

HLS allows for designs to be created and tested in languages like C by transforming high-level code into an RTL representation like VHDL or Verilog. This also allows for the opportunity for optimisations to be automatically performed between the transformation or user-specified directives which can influence the transformation to further improve or explore the resulting hardware implementation. This has significant TTM benefits alongside the ability to rapidly prototype and evaluate architectural decisions before commiting to a final design. HLS has a disadvantage in that it is only as good as the designer utilising it and it can be unclear what the tool is performing, the generated VHDL and Verilog are difficult for a human to read [35].

The tool used in this project; Vitis-HLS, Vivado and Vitis, alongside the hardware platform are from Xilinx. This thesis will now focus on Xilinx specifically, what the platform is and how Vitis-HLS can utilise its resources effectively through varying levels of abstraction.

2.8 Xilinx Tools for FPGA Development

Xilinx provides development tools for software and hardware development targeting their FPGAs and SoC platforms. These encompass different stages of FPGA development, including software development, toolchains, and customization of embedded Linux for their SoCs. The main focus of this report are the tools used for FPGA and software development.

Vitis-HLS

Vitis HLS is a tool provided by Xilinx for designing FPGA hardware implementations in high-level languages like C and C++. This operates as an Integrated Development Environment (IDE) which facilitates the conversion of algorithms into a high-level format that allows for the automatic generation of RTL for implementation on FPGAs. This tool supports the development flow by allowing for the creation and utilisation of high-level testbenches with breakpoint debugging, compilation tools, co-simulation, deadlock awareness and wave-form analysis. This allows designers to focus on development and performance optimisation [36].

Vivado

Vivado is an tool and FPGA design suite for creating, implementing and verifying FPGA designs. Vivado includes synthesis, placement, routing, debugging, and waveform analysis. Vivado also functions as an IDE for RTL development in VHDL/Verilog. Upon creation of an RTL IP or HDL design Vivado is necessary for the integration into an SoC platform which can be exported into Vitis for progamming or converted into a programmable bitstream [37].

Vitis

Vitis is an IDE for software and hardware development, with the aim of developing software to utilise accelerated applications. Vitis provides debugging in C-simulation, co-simulation and at the hardware level over JTAG for unified hardware-software applications. Vitis supports programming languages and frameworks, such as C, C++, OpenCL, and TensorFlow, making it accessible to a wide range of developers. Vitis also provides optimization and profiling tools to help achieve the high performance from accelerated applications [38].

2.9 Xilinx SoC System Organisation

This thesis focuses on the Xilinx MPSoC with FPGA platform, the ZCU-104 [39], HYPSO-1 uses the Zynq-7030 SoC. These systems are divided into a hard Processing-System (PS) and reconfigurable Progammable Logic (PL), see Fig 2.16, which are able to interact through a hard standardised Advanced-eXstensible-Interface (AXI) bus. AXI is part of the Advanced Micro-controller Bus Architecture (AMBA) open-standard by ARM.

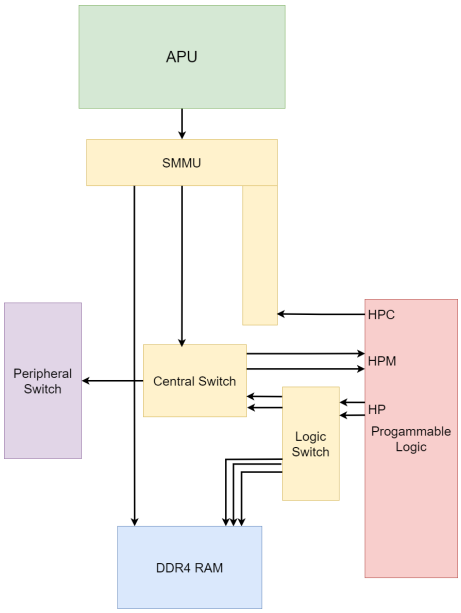


Figure 2.16: Block diagram of relevant SoC components [40].

2.9.1 Processing System

Application Processor

The primary application processor of the ZCU104 is the Arm Cortex-A53 MPCore. It consists of a cluster of four cores, each equipped with a floating-point unit (FPU), NEON, and Crypto computational units. Additionally, each core has a 32KB Level 1 shared instruction/data cache. The ZCU104 also features a shared Level 2 cache of 1MB, along with a snoop control unit, a master bus interface, and an accelerator coherency port. The purpose of the accelerator coherency port is to provide synchronization of cached data with any PL accelerator present in the system [40].

2.9.2 Programmable Logic

FPGA

The MPSoC incorporates the xczu7ev-ffvc1156-2-e FPGA. This FPGA offers a significant capacity with 504K+ Logic Cells, allowing for complex digital designs and computations. It operates at a clock frequency of up to 600MHz, providing fast processing capabilities. The FPGA also includes on-chip memory resources, such as 36Mb of on-chip UltraRAM and 36Mb of on-chip BRAM. These memory elements provide efficient dense on-chip storage for data, contributing to the flexibility in tailoring the FPGA to application-specific implementations. Additionally, the FPGA incorporates DSP48E2 slices, which are specialised digital signal processing units offering dedicated hardware resources for performing mathematical operations commonly found in signal processing applications [40].

2.9.3 Shared Memory and Interconnect

The PS of the ZCU104 is equipped with a level 3 memory system, which includes 8x256MB (2GB) DDR4 RAM modules. These RAM modules provide a total capacity of 2GB for storing data and instructions. The FPGA in the ZCU104 is also capable of accessing this DDR4 RAM. To facilitate this, AXI interfaces of the PL are connected to specific high-performance ports of the MPSoC. This connectivity allows the FPGA to communicate with and utilise the DDR4 RAM resources available in the system. Optionally, to ensure proper coordination and coherency between the FPGA and the CPU, both components share a connection to the System Memory Management Unit (SMMU). The SMMU plays a crucial role in managing memory accesses and ensuring cache consistency. In cases where the PL is utilizing the high-performance coherent (HPC) port, the FPGA accesses the DDR4 RAM through the SMMU. However, if the PL is not utilizing the HPC port, the FPGA can directly access the DDR4 RAM without involving the SMMU. General-purpose ports are also available for AXI-Lite register access from the PS to FPGA, enabling the processor to write and read the FPGA's internal registers using memory-mapped operations [40], [41].

2.9.4 AXI Bus

The AXI bus operates is a synchronous bus and is divided into masters and slaves. Masters are responsible for initiating data transfers, while slaves respond to these requests, see Fig. 2.17. Within the SoC, masters and slaves are organised through a centralised AXI interconnect matrix, which efficiently routes data to various bridges on the bus. This interconnect matrix forms a hierarchical structure, allowing for the connection of different peripherals or co-processors that may have varying speeds, see Fig 2.16 once again. As a multimaster bus, the AXI bus necessitates arbitration to handle situations where multiple masters attempt to control the same bus signals simultaneously. In such cases, an arbiter is employed to determine which master gains control of the bus through arbitration. Transfers involve a handshake mechanism between the master and slave. The typical handshake process includes the exchange of a valid signal and a ready signal. The valid signal, transmitted from the data source, indicates that valid data is ready to be transferred. The ready signal, sent by the data destination, indicates that it is prepared to read the data. When both the valid and ready signals are high simultaneously, it signifies that the data has been successfully read, enabling subsequent transactions to occur [42].

2.9.5 AXI-4 Bus Features

AXI-4 Bus Features

- Bus Locking
  The AXI bus incorporates bus locking functionality to facilitate atomic uninterruptible transfers. This feature is helpful for implementing semaphores and synchronization mechanisms.

- Concurrent transactions
  Masters on the AXI bus have the capability to initiate multiple transactions concurrently, and these transactions can be completed in an out-of-order fashion. This eliminates the requirement for a split bus command since the bus is not locked by a single transaction request.

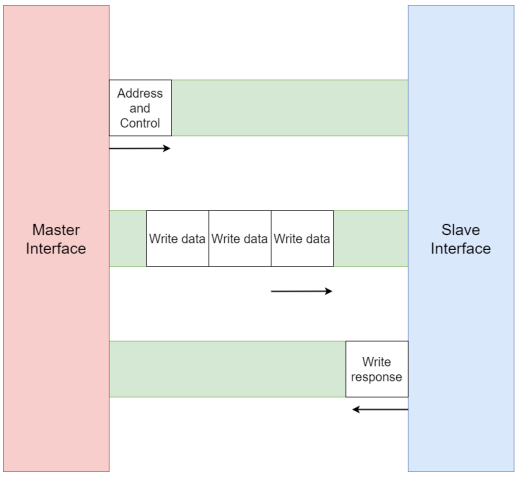


Figure 2.17: Multichannel AXI [42].

- Pipelined transactions
  When a slave acknowledges a transfer, even before it is ready to respond, another request can be initiated. This overlapping of the slave's response and the new request allows for a higher throughput on the bus. However, this approach requires dedicated acknowledgments for each slave device involved in the transactions.

- Burst Transfers
  Burst transfers offer the ability to initiate a transaction with a slave device in a single address phase. During this process, the base address is transmitted in the first phase, along with an additional address increment size. By holding the read or write signal high, the address is automatically incremented. This convenient auto-increment feature effectively doubles the bandwidth for large transactions, as there is no need to send separate address phases for each data transfer within the transaction.

- Coherency
  The AXI-4 bus standard provides options for coherency with the MMU of the SoC it is integrated into. This enables the bus to directly read an address from the cache, ensuring access to the most up-to-date value for the APU and chosen peripherals including FPGA.

2.9.6 AXI Interfaces

AXI4-Memory Mapped

The AXI4 protocol is designed for high-performance memory-mapped communication to and from the FPGA. It facilitates efficient data transfer between masters and slaves by supporting bursting, with a maximum limit of 256 data cycles. This capability enables higher throughputs for data transmission. AXI4 incorporates five channels: read address, write address, read data, write data, and write response. These channels allow for simultaneous reading and writing of data between the master and slave components. In AXI4, data sizes can vary and are defined during the instantiation of the Intellectual Property (IP) module. Vivado, a tool by Xilinx, supports data sizes ranging from 16 bits to 1024 bits. These interfaces also support data buffering when used off-chip or between different clock domains, enhancing their flexibility. In scenarios where an IP module has an AXI4 interface with a width of 512 bits and is implemented at a clock frequency of 100 MHz, but the bus data width only supports 128 bits, the IP module can still receive all 512 bits of data within one cycle if the bus has a high enough clock frequency of [43].

AXI4-Lite

AXI4-Lite is a simplified version of the AXI4 protocol, specifically designed for low-throughput memory-mapped communication. It is well-suited for handling control and status registers. Similar to AXI4, AXI4-Lite is a memory-mapped interface, but it lacks support for burst transfers [43].

AXI-4Stream

AXI4-Stream is a unidirectional, point to point channel that is derived from the data write channel of AXI4. It is specifically designed for data-flow-based designs. Unlike other AXI interfaces, AXI4-Stream does not require the forwarding of an address during transfer as it is not memory mapped. It supports an unlimited burst size, making it suitable for scenarios where continuous data streaming is required from component to component within the FPGA [43].

Chapter 3

Literature Review

This literature review section will provide an overview of the general techniques used to create optimised hardware accelerators targeting the domains of image processing and AI. This is done through analysing examples in literature that cover FPGA accelerators for image processing and for AI applications and includes system architectures and the concerns regarding floating point implementations in FPGAs.

3.1 FPGAs Accelerators

3.1.1 FPGA Image Processing

FPGA-based image processing involves the transfer of image data onto an FPGA for processing, often within an image processing pipeline. The selected architecture and design choices significantly impact resource utilisation and throughput. This directly influences the ability to meet operational requirements efficiently without compromising performance or necessitating excessive resource allocation and increasing power consumption. There are common principles followed in most designs, one significant similarity in many FPGA designs is the utilisation of on-chip BRAM memory as a cache or buffer to store a portion of the images during processing. This approach commonly involves a line buffer strategy with a sliding feature map, enabling a representation of the 2D space that matches the size of the applied mask without the need to buffer the image outside of the lines being processed [44][45]. In Angelopoulou and Bouganis [44] which implements vision-based egomotion estimation, the FPGA is predominantly utilised for critical path processing by applying feature maps and feature selection. Resource hungry operations that are less critical are offloaded to the CPU using a hardware/software partitioning approach. To support more advanced buffering strategies, external memory is utilised alongside the FPGA as a form of multi-level cache. To make full use of the RAM bandwidth a zig-zag access pattern is used from off-chip memory in Fig. 3.1. The distribution of image data among 4 RAMs allows for quadruple the bandwidth but requiring a more complex memory address generation. This alleviates some on-chip memory requirement as RAM is the densest memory available on an embedded SoC.

In Ulusel et al which creates an architecture for applying a Bresenham circle mask [45], image frames are retrieved from RAM and read into on-chip line buffers. The reading from RAM is similarly in a zig-zag pattern to maximise re-use in the line buffers. Each line buffer stores a row of image data and are accessed by pointers. The first line buffer is fed pixels from memory and feeds into the feature mask and subsequent line buffers, such that data is shifting from buffer to buffer and the oldest data is overwritten by the newest, this is visualised in Fig. 3.2. Data is written from these line buffers into a fast shift register array cache that is used in applying the feature mask. The computational speed of processing is determined by the time required to place pixels corresponding to a single column of the mask into the register array cache rather than filling the entire array as a result of this shifting. The array has dimensions of 7x7 and so this approach processes and shifts the array horizontally, reusing the subsequent columns.

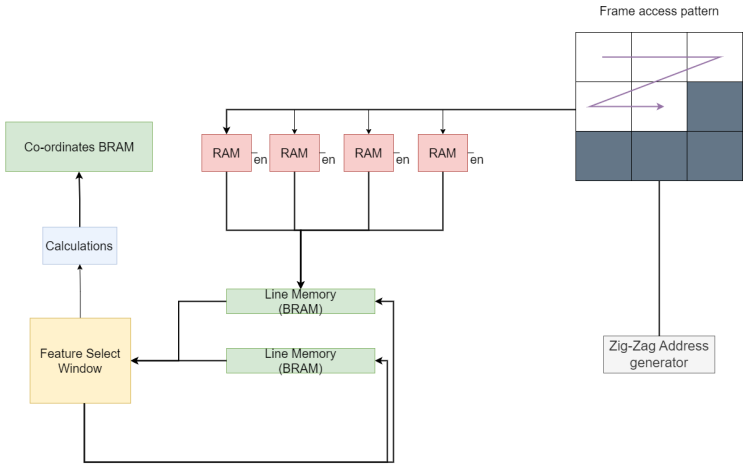


Figure 3.1: Memory access pattern and caching of image data from Angelopoulou and Bouganis [44].

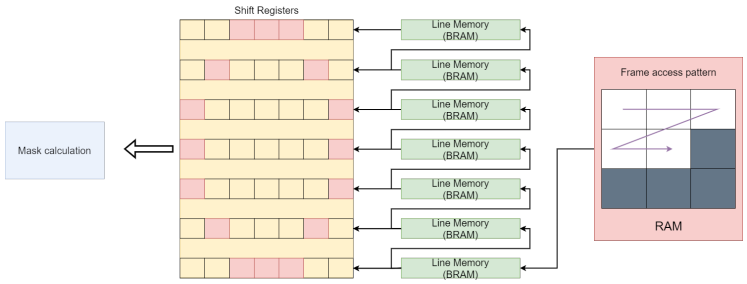


Figure 3.2: Memory access pattern and caching of image data from Ulusel et al. [45].

By minimising the necessary transactions with off-chip memory and an inventive organisation of on-chip memory, these accelerators demonstrate high performance through efficient on-chip caching strategies. This literature highlights that utilising on-chip memory is going to be a key part of the implementation of this project for both the AE accelerator and especially important for the DSW accelerator. There are layers of memory available on the MPSoC, including main SoC RAM, and within the FPGA, on-chip BRAMs, URAMs and fast FFs and LUTs, and a synergy of these components will need to be found for an optimal implementation.

3.1.2 Architectures for AEs

DNNs typically involve a large amount of MAC operations performed continuously over a large set of data, AEs are no exception. The most challenging part is the management of on-chip and off-chip weights and biases together with reading input data for scheduling these MAC operations. Accessing all of this image, weight and bias data repeatedly from memory becomes the largest source of latency and power draw [26]. Strategies to mitigate this issue try to maximise on-chip re-use of parameters, input and intermediate outputs of calculations. For an AE as a combination of FC layers this means a strategy for high re-use of weights, biases and input bands.

DNN accelerators typically consist of off-chip memory, on-chip buffers and a network of processing elements (PEs). This review focuses on FC layers in particular, these are used in the AE for performing DBN based anomaly detection as RBM1 and RBM2. Which compose a majority of the processing. These act as a matrix-vector multiplication and have the opportunity for input re-use. This is because a single band input is used across the MACs for every mid layer code, so once it is used in a MAC operation for a singe code it can be moved to the next mid layer code calculation. This re-use creates an incentive for and so is commonly combined with weight-stationary architectures. Stationary meaning they have a fixed weight and PE organisation such that input data is re-used as it moves through an array of PEs with a fixed weight input for multiplication and is subsequently accumulated [26]. The AE architecture presented in the background and to be

accelerated would only have a small array of weights and parameters as it is encoded and decoded pixel-by-pixel based on FC layers encoding and decoding the number of bands, which means the organisation of PEs for a weight stationary architecture would be similar to as in Fig. 3.3.

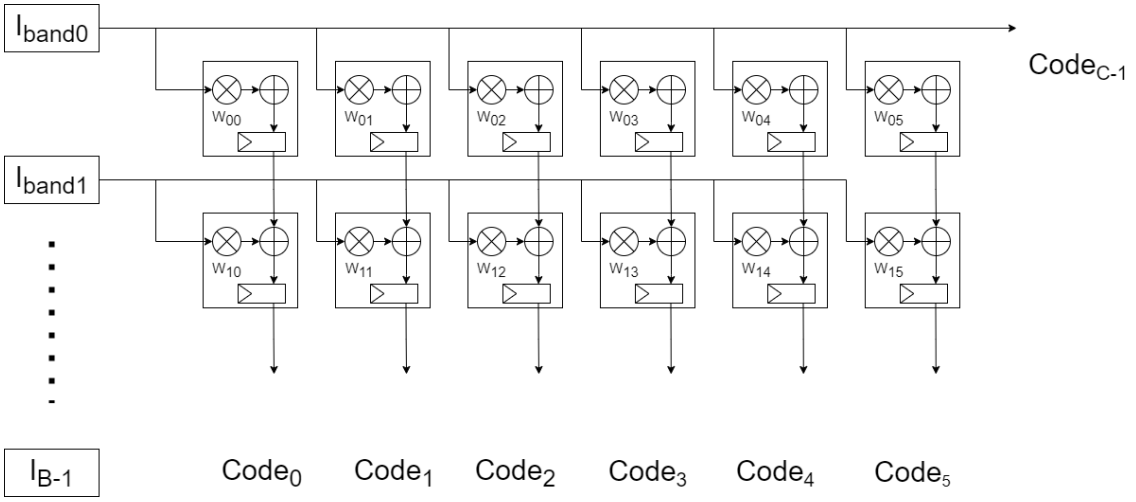


Figure 3.3: Adaptation of weight-stationary input-reuse accelerator architecture for multiple band FC on encoder [26].

Figure 3.3 provides an example of a weight stationary network which broadcasts every input among all of the hidden code node MACs as a form of re-use. After this broadcast the MAC is performed and the result is re-used in the next inputs MAC operation. If intermediary registers where added between input stages this could also be a pipelined architecture.

3.1.3 Floating Point

Much literature around FPGA implementations focus on the change from floating point to fixed point data representations. The reason for this is that floating point numbers are stored in signed-magnitude format, in combination an exponent field to control the position of the decimal point. IEEE format also specifies adding an exponent to the floating point bias, alongside an implied 1 at the point. There is a significant difference in all operations when implemented in floating point compared to integers. A single floating point operation can consist of 30 or more integer operations [33]. As an example see the floating point multiplication implementation targeting DSP48E slices in Figure 3.4. A single multiplication requires three simultaneous computations and three processing stages.

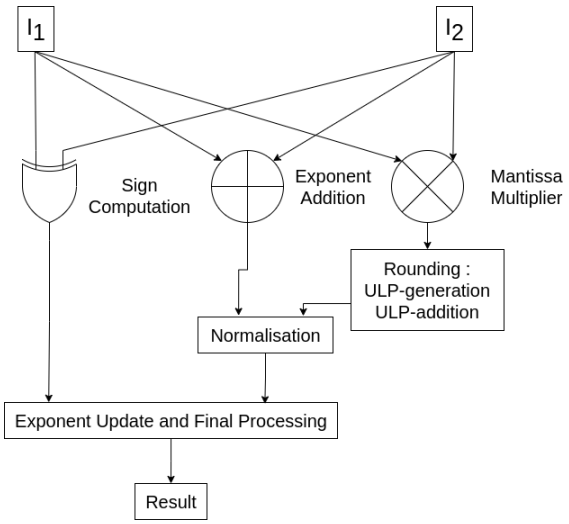


Figure 3.4: Floating point multiplication from [46].

3.1.4 Fixed Point and Quantisation

While floating point numbers are highly flexible with a large dynamic range the price in terms of hardware resources becomes highly prohibitive. Alternatively, implementations with conversion to fixed point have been shown to reduce DSP, FF, LUT utilisation, increase power efficiency and improve latency [47].

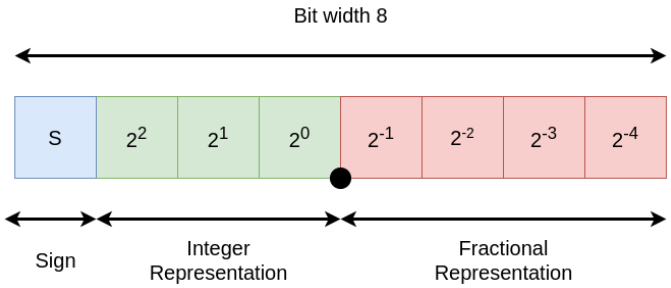


Figure 3.5: Fixed point data representation, as seen in [48].

Fixed point has an exponent field that cannot vary with time. This fixing eliminates the need for alignment and removes the shift requirement associated with floating point, see Fig. 3.5. This is desirable in reconfigurable architectures. In [48] research was performed into fixed point quantisation for CNNs specifically, and layer-wise quantisation where each layer of the NN has a different parameter for its quantisation was found to have improve area saving and performance improvements without significantly impacting accuracy compared to a single quantisation over the entire network. However, accommodating a fixed point representation in a design comes with three challenges [33].

1. Determining a peak-value estimation to eliminate the possibility of the fixed-word overflowing or using a saturation strategy.
2. Testing or analysis for an acceptable precision in fixed point to prevent excessive accuracy degradation through rounding.
3. Finding a suitable variation of 1 and 2 for all computational tasks throughout the architecture.

Peak value estimation

The intention of peak value estimation is to prevent overflows, to ensure the circuit or architecture does not change functionality through the change to a fixed point representation. There are several approaches, simulation based and analytical based approaches to finding a peak size. A Simple simulation based approaches are to measure the peak achievable values of signals in the system [33].

Wordlength Optimisation

The purpose of wordlength optimisation is to find a optimal fractional length for preserving the required precision. This means trading area for accuracy. This can be performed by finding an average error over potential fractional lengths and choosing the most optimal length with an acceptable accuracy loss [33].

Quantisation

Quantisation of NNs involves modifying or training an NN to operate in a fixed point representation. This has benefits for both the software and significant benefits for the FPGA implementation. If the training has also been fine-tuned or performed for fix-point representations this will allow for the accuracy to remain near-ideal relative to the floating point implementation [48].

Chapter 4

Implementation

4.1 Model Methodology

Before commencing the implementation stage of the project a review was undertaken of prior work carried out by Gunderson [3] on AWDBN and its adaptation into the novel weighting strategy. This section will first present an analysis of the current high-level Matlab solution for anomaly detection discussed in the literature review. Secondly it presents changes to the high level code with justification for these changes and finally introduce the reference model that will be used to validate the hardware implementation.

The suitability of high-level Matlab code for the task of anomaly-detection will be assessed along with its potential adaptability into a hardware accelerator. Any necessary changes made will be justified and documented. The focus of the analysis is identifying potential limitations and proposing adaptations in order to achieve a high-speed implementation while maintaining near-ideal accuracy. As part of the investigation the best parameters for implementation will be identified and used for the reference design.

This section also aims to inform the reader of the validation strategy. This is through justifying which training algorithm and parameters are picked for the reference model of Matlab. Using these trained weights and parameters, a high-level C implementation will be written that will be used to check replicability of the Matlab results and used continuously for validation over the development of the hardware accelerator in HLS.

Parameter	**Network Architecture**
Bands	Input and output layer nodes
Code Size	Hidden layer nodes
	AWDBN
Win Size	Inner window length
Win Dif	Outer window difference
Penalty Threshold	Deviation from mean required to be penalised
Penalty Factor	Penalty multiplication
	Network training
Batch Size	Samples executed in NN before updating neurons
Iterations	Batch forward passes
Step ratio	Rate at which weights are updated

Table 4.1: Network, training and algorithm parameters

4.1.1 Suitability of Prior Work

The code from Gunderson's thesis [3] was studied in detail, this code implements a DBN AE with one input layer one hidden code layer and one visible output layer, before calculating reconstruction errors and applying the adaptive weights strategy. This architecture is based on the input band size matching the output size and the middle code layer size being customisable. The most significant parameters are indicated in Table 4.1. These are the parameters that were optimised

in Gunderson's thesis.

Some slight variations from the original AWDBN algorithm proposed by Ning et al were identified. These variations were seen in Gunderson's implementation of AWDBN and were subsequently carried into the Novel method used. Gunderson also implemented a HLS hardware version in C of the HSI-AE inference which has a slight variation from the Matlab code for the same inference.

Coherency of literature and models

To create a consistency between the literature, the high-level model and what was implemented in HLS by Gunderson, the following changes were made.

- At the end of the reconstruction error calculation included in Gunderson's Matlab code, which is the euclidean distance between the input and output pixels, there is a division by the number of bands which is not present in the original algorithm by Ning et al. After removing this there was no observed difference in detection accuracy and this change means the hardware implementation will use less resources, so it is a beneficial change which will be carried forward to the implementation stage.

- A change was made to the code distance calculation. In Gunderson's Matlab code the code distance calculation varies from Ning et al. [28] as instead of performing a euclidean distance between the PUT and code pixel, the absolute difference between each code band is summed and square rooted, rather than root sum of squared differences. This was changed back to what was provided in Ning et al. This change came in as slightly detrimental to the AUC score when considering the best window for multiple window parameters for each image. However, when considering only single fixed window parameters for all images this change had a slight AUC improvement. As this project aims to implement only single fixed window parameters the change to Gunderson's code is chosen.

- There was an error in the applying of penalties in the Matlab code for the AWDBN and novel AWDBN. The penalty is applied if the reconstruction error varies from the mean by a standard deviation. In Gunderson's code the calculation was; if the distance of the reconstruction error of a pixel was above the neighbourhood mean by a standard deviation a penalty factor is applied. This was improved by applying an abs() to the calculation, taking absolute distance from the mean which improved the AUC slightly.

- When evaluating both of the algorithms with AUC, the Novel-AWDBN algorithm searched through every possible window variation and picked the best by brute force. This search was only done for the Novel-AWDBN and not the regular AWDBN and the window size was applied to both without distinction. This previously assumed the best window for the Novel would be the best for the AWDBN, which may not always be the case. Now the code searches for the best possible window for both individually and as a result this improved the best-case AUC for the AWDBN.

Now, with the formation of a high-level model consistent with background literature, the investigation of necessary adaptations for transforming this into a hardware implementable solution will be presented.

Improvement candidates

There is an issue in the current Matlab implementation regarding the sliding window, see in Fig. 4.1. The edges around the image are dark and not evaluated in either AUC or given a reconstruction error. This is because the sliding window is not applied to all the pixels of the image, rather it is applied from the first pixel the window can surround without crossing out of the image boundary see in Fig. 4.2. This means that the number of pixels evaluated changes based on the size of the window. Additionally, this means that the best window picked may be chosen not because it is the most accurate window but because it skips troublesome pixels or even anomalies.

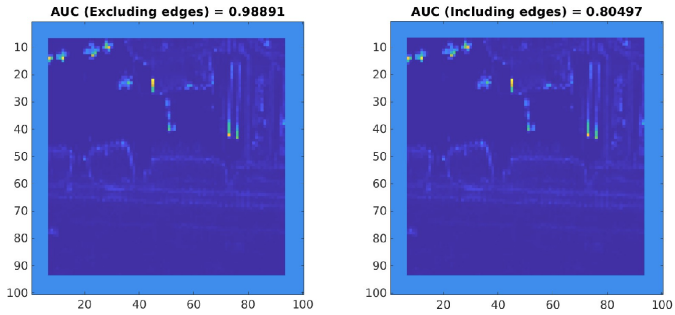


Figure 4.1: Problem in original algorithm; image boundary skipped and evaluated without skipped area

To solve this problem two possible solutions were considered and compared to the Novel-AWDBN.

1. **Original**; If none of the above solutions are acceptable, the original code will be implemented.
2. **Fill**; As this original implementation in this form does not worry about padding or only partially full windows, it is very fast. While it is not acceptable to miss parts of the image, a minor change to the algorithm is to return the original reconstruction error from the autoencoder for the pixels that the window cannot fit around as the anomaly score for that pixel.
3. **Edge**; The other solution is to have the window traverse the boundary of the image, by either using padding to extend the image or skipping out of bound values by preventing the index out of bound pixels. This effectively means the window is limited to a smaller size at the corners and borders of the image, at the top left corner the window would go down to as much as a quarter of its original size. This means that the window will be less robust than what was originally proposed in Ning et al. as it performs its average and standard deviation over a smaller sample of pixels. However, it may be better than the Fill proposal in terms of detection performance because it still has a spatial component. A concern would be anomalies on the edge of the image that may be large enough to encompass the entire window.

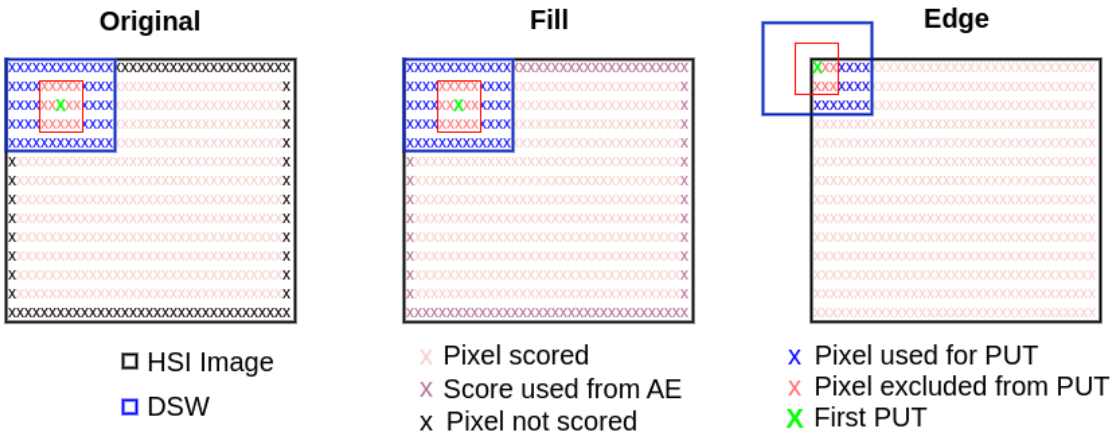


Figure 4.2: Overview of window algorithm candidates

Candidate evaluation and performance

The Edge candidate has a better AUC than the Fill. The Edge candidate is only slightly lower in AUC than the Original.

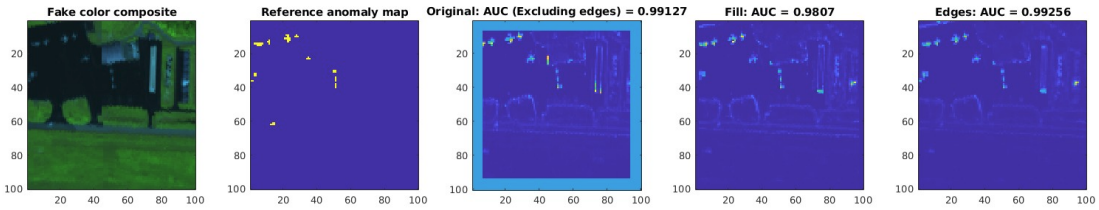


Figure 4.3: Comparison of window algorithm candidates on urban 3

Accelerator suitability

- The Fill and Original candidates would be almost identical in terms of a hardware implementation. Both would use the same on-chip memory organisation, the only difference would be that the reconstruction error would be sent as the anomaly score directly when the window is not fully in the image boundary.
- The Edge candidate requires more complex logic. A padding strategy would need to be implemented for the padded version. For the non-padded version every cell of the window would require a comparison to see if they are still within the image. The implications this has on reducing speed could be mitigated through unrolling and pipelining on the FPGA at the cost of hardware resources.

Candidate selection

As the Edge candidate provides the most coverage over an image with near-ideal AUC scores, this will be the chosen algorithm for the hardware implementation. This is because regardless of interpretation by HLS and choice of how this accelerator would be integrated into the SoC, such as a direct connection from the AE unit to the DSW unit or individual peripheral units both accessing DDR4 RAM, all three implementations should have satisfactory performance in terms of speed. The specifics of the hardware implementation for HLS will be addressed in the implementation section for the double-sliding window. Any implementation of the Edge candidate should perform the same in detection if given the same input and operating parameters. These parameters include the size of the DSW, training algorithm and training parameters used to generate the weights, biases and output used in testing. For this reason these parameters will be decided on and a model will be created for use in HLS.

Images	**Scores (No Boundary Edges)**		**Best Window Paremeters**		**Score (With Ignored Edges)**		**Scores (Chosen Edge Candidate)**		**Best Window Parameters**	
Dataset	AWDBN	Novel AWDBN	AWDBN	Novel AWDBN	AWDBN	Novel AWDBN	AWDBN	Novel AWDBN	AWDBN	Novel AWDBN
Air1	0.907079	0.966729	3, 4	1, 6	0.777027	0.790198	0.922589	0.965833	1,6	1,6
Air2	0.949404	0.975091	3,6	3,6	0.9648	0.983402	0.925598	0.97174	1,5	1,5
Air3	0.915997	0.973371	5,6	5,6	0.928609	0.951232	0.910084	0.969645	7,6	7,6
Air4	0.943938	0.964453	3,2	3,2	0.955345	0.967786	0.95809	0.966349	1,4	1,4
Air average	0.929105	0.969911			0.90644525	0.9231545	0.92909	0.968392		
Beach1	0.999688	0.99031	3,1	3,1	0.999673	0.997696	0.99962	0.997785	3,1	3,1
Beach2	0.996032	0.99031	1,6	1,4	0.99659	0.992357	0.99424	0.987088	1,2	1,2
Beach3	1	0.999987	3,1	7,4	1	0.999991	1	0,999968	3,1	7,1
Beach4	0.979329	0.993318	15,3	15,3	0.865379	0.87998	0.95704	0.984573	1,6	1,6
Beach average	0.993762	0.993481			0.9654105	0.967506	0.987725	0.992354		
Urban1	0.989251	0.997082	5,1	1,6	0.917569	0.932318	0.987549	0.996947	1,3	1,3
Urban2	0.971662	0.99483	15,6	13,6	0.986326	0.997291	0.942126	0.995776	15,6	15,6
Urban3	0.987049	0.990985	13,1	1,6	0.787409	0.830931	0.987239	0.992394	9,3	1,3
Urban4	0.985811	0.992008	9,4	9,6	0.714046	0.669268	0.977611	0.99241	5,6	5,5
Urban5	0.937522	0.96708	7,2	7,1	0.904466	0.954337	0.941206	0.971164	5,3	5,2
Urban Average	0.974259	0.988397			0.8619632	0.876829	0.967146	0.989738		
Overall Average	0.966366	0.984845			0.90748	0.918984	0.961769	0.983975		

Table 4.2: AUC of selected edge candidate versus original with best window sizes.

Parameters for implementation and evaluation

Hardware accelerators lack the flexibility of software, so some parameters and test datasets need to be fixed to a constant and compared to a matching reference model. This is for hardware implementations to have their output verified by comparison to the model to verify that the implementation is performing its algorithm correctly.

Gunderson's thesis contained two methods for training the AE. The method in C was shown to have reduced accuracy compared to the high-level Matlab training when using the AE for anomaly detection directly. It is suggested this difference is due to a lack of pre-training of the AE. As the more accurate model the Matlab trained AE will be used for the evaluation model. It will be used to create weights, parameters and, together with the Novel-AWDBN with edges, the expected output for every HSI. The training parameters used in the Matlab code will be used from Gunderson's thesis directly. This thesis contained considerable effort for finding the best training parameters for the HSI AE in Matlab. These parameters are as shown in table 4.3.

Parameter	**Network Architecture**
Bands	19 to 207
Code Size	6 to 13
	AWDBN
Win Size	Set of windows
Win Dif	Set of windows
Penalty Threshold	Greater than one standard deviation from mean
Penalty Factor	0
	Network pre-training
Batch Size	7
Iterations	15
Step ratio	0.02
	Network training
Iterations	40
Step ratio	0.01

Table 4.3: Table of model reference parameters.

The last parameters to choose are those for the size of the DSW and penalty factor. Gunderson stated the best observed penalty factor is 0, meaning that any reconstruction error that varied from the mean was effectively removed from the anomaly score calculation. No value has been found so far to contradict this statement. A hard-coded penalty factor of 0 in the hardware accelerator could be particularly beneficial for creating a fast implementation, as a result this parameter is chosen to be 0, again in 4.3.

The DSW, which was tested in multiple configurations, has no heuristic for finding a best window size, the best size was selected by testing every size within a range and picking the one with the highest AUC score. Creating a hardware implementation of the DSW that can perform for many sizes would require a large amount of resources on the FPGA for a minor gain in AUC. Futhermore having to find the best size by brute-force would defeat the point of acceleration. As a compromise, a single window size is picked for synthesis but the design can be re-synthesised for any size of window. The reference model will be based on the best size parameters from an average over every image explored for 10 iterations. The graph of AUC and parameters can be seen in Fig. 4.4. The best size average has an AUC of 9.795, an inner-window of size 1, with an outer difference of size 5, corresponding to the green line. This creates a total window of 11 by 11 pixels. This window has a large processing requirement due to having 120 neighbours and so a selection of the best windows for different processing requirements will be additionally explored. These windows are displayed in Table 4.4.

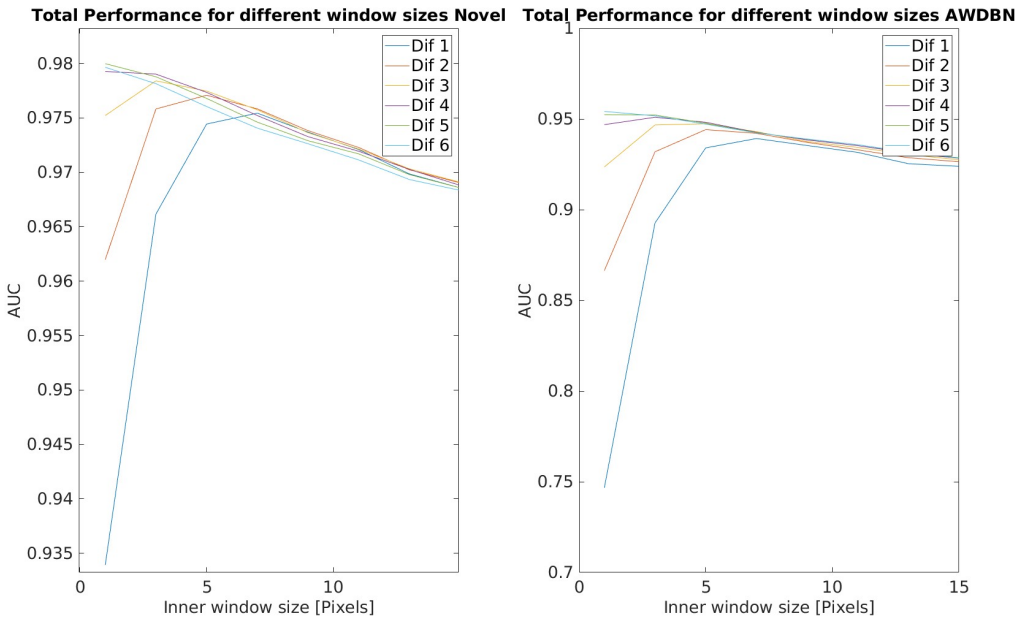


Figure 4.4: Average AUC for all window variations of 1x1 up to inner window size 15x15 and an outer difference up to 6 pixels for the Novel-AWDBN.

AUC (Avg)	**Inner Size**	**Outer Difference**	***Window Size***	***Neighbours***
∼.98	1	5	11x11	120
∼.979	1	4	9x9	80
∼.978	3	3	9x9	72
∼.976	2	5	9x9	56
∼.975	7	1	9x9	32

Table 4.4: Table of accuracy, window size and number of neighbours that contribute to the PUT anomaly score.

4.1.2 Evaluation Strategy

The reference model is essentially the exported weights, biases and results from running the Matlab code with the established parameters. The average AUC of this configuration on ABU is 0.980. As part of the HLS flow for hardware accelerator development a refined C model that performs the HSI-AD anomaly detector tasks of inference of AE and Novel-AWDBN has been created which covers the edges in the described way. From now on this will be what is referred to as the Novel-AWDBN

4.1.3 Evaluation Model

A refined C model has been created to run in HLS, to verify the hardware implementation. This evaluation model has been verified by comparison with the Matlab results for exported weights, biases, and desired output with the parameters set as described. These will be used in testing in various stages throughout the accelerator development, including software-simulation, RTL co-simulation and the final implementation running on the FPGA. After testing the C implementation for every image, the AUC matched at 0.980 with a deviation in results by 2%. This deviation may be due to rounding errors as the C implementation uses single precision floating point types.

4.2 Autoencoder Exploration Base Architectures

This section aims to provide the reader with an understanding of the starting point of the hardware accelerator of the autoencoder portion of the HSI-AD. In addition it will introduce potential optimisations and candidate architectures which can be further improved by the use of directives in the design exploration. Initially analysis will focus on PCA 32-Band 40x40 pixel HSIs and then modifications will be included for larger band sizes.

The initial architecture comes directly from Gunderson's C code for AE, which was also implemented using Vitis-HLS by Gunderson in a more limited scope [3]. Using the C code directly ported to HLS as a basis for further implementations the following architectures will be presented in this section;

1. Baseline
2. Baseline with On-chip Weigths and Biases
3. Dataflow
4. Pipelined
5. Reordered
6. Retiled

Additionally terms used to asses performance of the hardware architectures are defined in Table 4.5.

Term	Definition
Initiation Interval (II)	Delay between processing subsequent pixels
Latency	Clock cycles required to complete execution
Throughput	Inputs accepted per second (Frequency / II)
Directive	HLS pragma which performs automatic transformations of C code.

Table 4.5: Terms used in accessing performance of hardware architectures.

4.2.1 Baseline Architecture

The starting point of this HLS implementation is porting the C code into Vitis-HLS. This porting is done by replacing the encoding function in SW with a function that is now linked to the HW for simulation and analytical purposes. HLS typically performs automatic optimisations, including partitioning memory for higher internal bandwidths and unrolling loops automatically for faster processing. In order to make it clear where opportunities for optimisations are and how this affects the resource utilisation, automatic optimisations are to be switched off through the following commands:

config_array_partition -complete_threshold 0
config_array_partition -throughput_driven off
config_compile -pipeline_loops 0

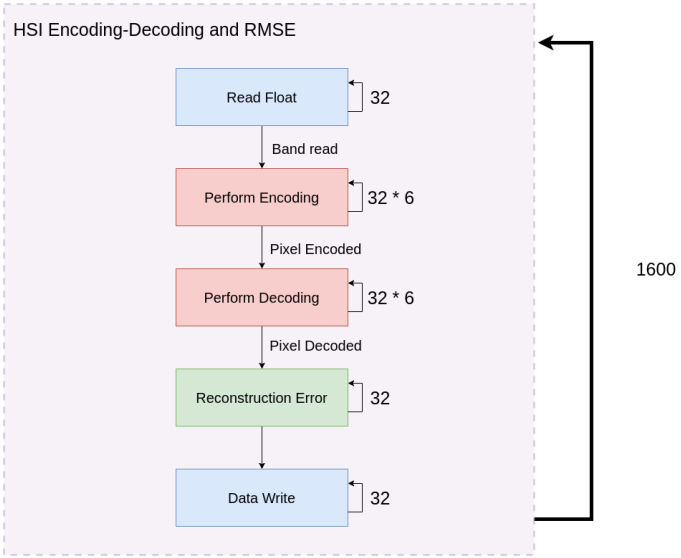


Figure 4.5: DFG of HSI autoencoder with band size 32, mid layer size 6 of width 40 and height 40

Analysis

The C code can be broken down into a simple data-flow diagram with loops and no feedback, see Fig 4.5. This entails the scope of the computations that are performed in the hardware accelerator. As a DFG of the problem overall, the C implementation does vary slightly from this DFG. The reconstruction error is coupled in the same outer loop as the decoding as a merged loop. This was a software optimisation. The initial design has CUs RBM1, and RBM2 with RE which contain sub-loops RBM1-Interior and RBM2-Interior respectively.

After synthesis of the design in Vitis-HLS a breakdown of the latency of each sub-module together with initiation interval is provided by the tool. Each task is running sequentially in hardware without any optimisations at this stage, see the results in table 4.6. In the table it should be noted that the initiation interval is equal to the sum of the latency of every function performed. This is because in this unoptimised design the initiation interval is equivalent to these functions running sequentially. Furthermore, the initiation interval multiplied by 40x40 is the total latency of the accelerator for a 32-band 40x40 pixel AE. How to arrive at the latency for a single pixel is demonstrated in Fig. 4.9.

Task	**Latency (Avg)**	**Initiation interval (Avg)**
Encode-Decode RMSE	20860804	-
Read Band	98	13038
RBM1-Exterior	5485	13038
RBM1-Interior	865	2172
RBM2-Exterior & RE	7425	13038
RBM2-Interior	163	407
Sqrt and Write	30	13038

Table 4.6: Table of characteristics of un-optimised basic implementation.

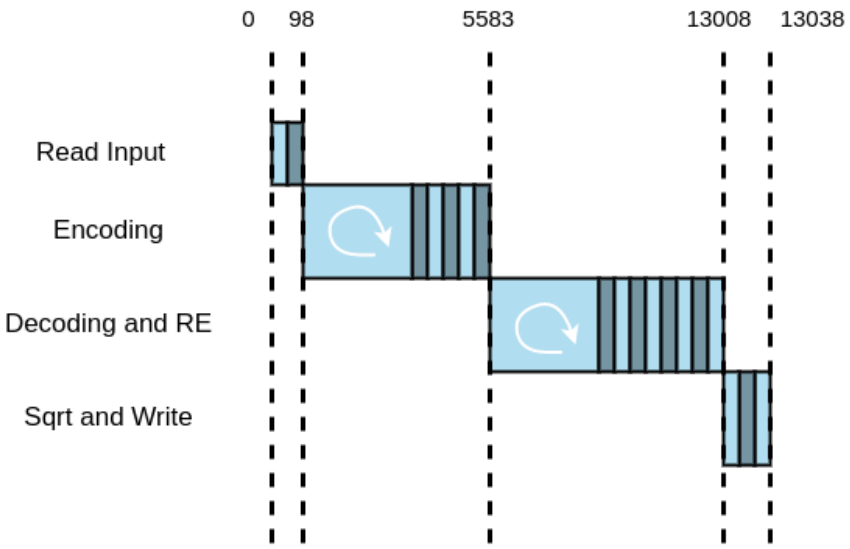


Figure 4.6: Graph of single pixel processed through initial design without concurrency or unrolled loops.

From the extracted DFG paired with insights of co-simulation on Vitis-HLS it is possible to understand the performance metrics of the design and their derivation. The design will be broken down and these metrics gathered from this point on-wards. Justifications for design changes will be provided, first in a task-by-task basis through the analysis of the initial design, and then all-encapsulating in the form of dataflow and pipelined architectures.

Reading Data

Figure 4.7 presents a micro-architecture breakdown of the first task of the HSI-AD, this will be used in tandem with Vitis-HLS co-simulation results to make observations about the architecture and propose improvements.

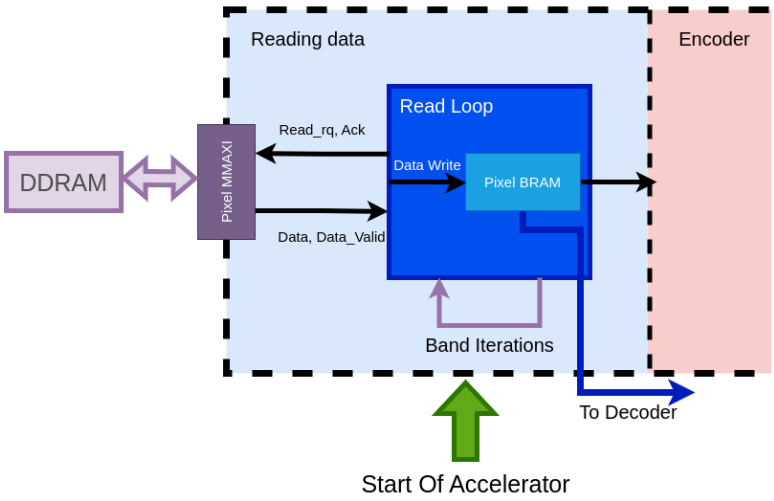


Figure 4.7: Read task of HSI autoencoder.

Read Submodule	Loop Initiation Interval	Latency	Trip Count	Total Latency (98)
Read Loop	**3**	**2**	**32**	**96**

Table 4.7: Table of characteristics of un-optimised basic implementation of read task.

The read loop performs the read requests over the MM-AXI interface to DDRAM. In co-simulation a read can be completed in one clock cycle and then written to the BRAM in a subsequent cycle, however there is a loop overhead and address calculation. This can be seen by the component having an iteration latency of 2 clock cycles. However the initiation interval is 3 clock cycles so every loop takes on average 3 clock cycles, combining with additional overheads to average II of 98 for this component.

To improve this pipelining the loop could be introduced. This would mean that while a read is occuring the next address is being calculated which would reduce the read latency and be used to lower the initiation interval if the rest of the design were to be pipelined. Another suggestion is instead of having a single pixel-BRAM a ping-pong buffer strategy could be used. This means that the pixel buffer can be written to and once it is in use and being read from a second buffer can be written to. This would be designed such that which one is being read by the AE at a specific time is alternated between each buffer as they are filled and emptied, eliminating the read overhead.

Encoding

The encoder module in Fig. 4.8 uses the same BRAM written to in the earlier reading task for its pixel data. This writes its intermediate and final results in BRAM memory for its MAC operations which is the hidden code layer H1. These weights and biases are read from an AXI-MM from DDRAM as they are used without re-use. In Table 4.8 it can be seen that RBM1-Interior, the multiply-accumulate is dominant in the latency. It has a latency of 865 and is performed 6 times.

The clear improvements seen in Table 4.9 come in the form of eliminating off-chip memory accessed in MAC by adding weight and bias re-use through on-chip memory. This should reduce RBM1-Interior from taking up 94% of the latency and reduce overall latency by 30%. This memory could be loaded in all at once before beginning the accelerator as the network is small enough to facilitate this. If these tasks were executed in parallel an optimisation would be similar to the reading of data, a ping-pong buffer to separate the current workspace H1 memory with what is being used in the following task of Decoding and RE. There are also opportunities to unroll or pipeline the RBM1 inner and outer loops at the cost of hardware memory and processing resources.

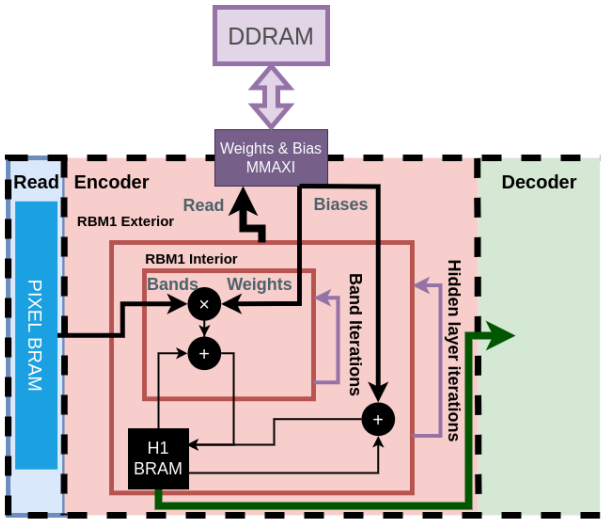


Figure 4.8: Encode task of HSI autoencoder.

Encode Submodule	Loop II	Latency	Trip Count	Total Latency (5485)
RBM1 Exterior (Including Interior)	914	914	6	5485
RBM1 Exterior (Excluding Interior)	914	49	6	295
> 6 RBM1 Interior	27	27	32	865

Table 4.8: Table of characteristics of a co-simulation of the unoptimised basic implementation.

Encode Submodule	Off-chip Memory	On-Chip Memory	Processing	Control / Other	Total Latency	Percentage of Task
RBM1 Exterior (Incl Interior)	225	33	393	72	754	100%
RBM1 Exterior (Excl Interior)	1	1	41	8	50	6%
> 32 RBM1 Interior	7	1	11	2	22	94%

Table 4.9: Schedule viewer table.

Decoding and RE

The decoder as seen in Table 4.9 uses the hidden code layer, H1 from the previous task. This similarly has the fully-connected layer implemented through MAC operations but with the outer-loop of RBM2 having more iterations than its inner-loop. Using the results of the co-simulation in Table 4.10 it can be seen there is a higher latency overall than the previous RBM1, this is to be expected due to there being more performed, notably the RE calculation. This coupling is also expected to increase on-chip memory access times as the decoded and working data is stored in the H2 BRAM and again accessed in the RE calculation. This process would ideally use a separate BRAM through memory partitioning to increase on-chip bandwidth.

As H2 is forwarded to an output and square root module, this could also benefit from ping-pong buffering. Being another fully-connected layer it similarly has a large latency cost from loading weights and biases, see Table 4.11. These should also be stored on-chip for future designs. The outer loops and inner loops can be unrolled or pipelined to reduce latency with a similar cost. As this task is currently the longest it could be preferable to separate these modules in a pipelined or dataflow architecture to ensure each module has a similar latency and initiation interval to reduce potential stalls.

Decoding and RE Submodule	Loop II	Latency	Trip Count	Total Latency (7425)
RBM2 Exterior and RE (Incl Interior)	232	232	32	7425
RBM2 Exterior and RE (Excl Interior)	232	70	32	2209
>32 RBM2 Interior	27	27	6	163

Table 4.10: Decode task characteristics.

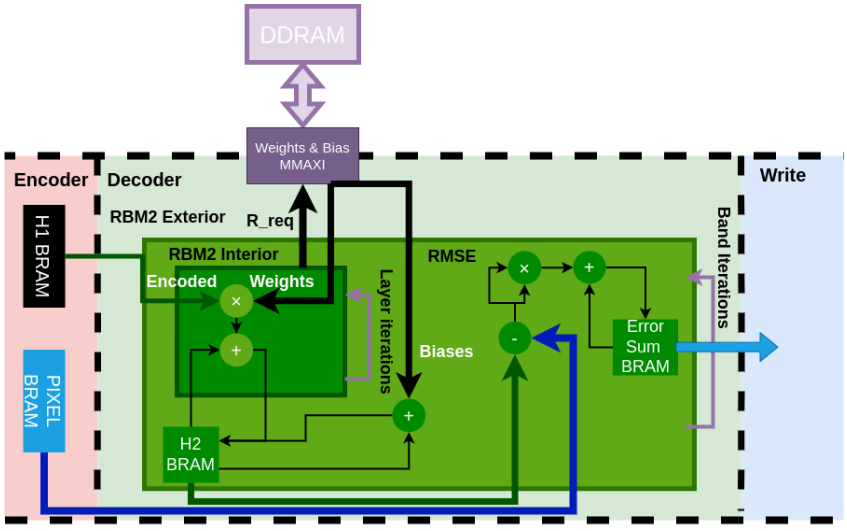


Figure 4.9: Decode task of HSI autoencoder.

Using the schedule viewer it is possible to generate an example of a best-case scenario which identifies where the clock cycles in the design are coming from. This information provides direction for considering what optimisations can be applied. From the table it can be seen that the internal iteration which has external memory accesses still provides a large latency. RME calculations also create a larger latency. These are therefore areas where improvements could be made.

Decoding and RE	Off-chip Memory	On-chip memory	Calculations	Control	Total Latency	Percentage of task
RBM2 Exterior and RE (Incl Interior)	43	6	134	14	202	100%
RBM2 Exterior and RE (Excl Interior)	1	0	68	2	70	35%
>32 RBM2 Interior	7	1	11	2	22	65%

Table 4.11: Decode task schedule

Writing data

The module can be seen in Fig. 4.10, there are no loops as a single reconstruction error is transferred from this module after a square root is applied. From the co-simulation results in Table 4.12 it can be seen that the overall latency is 30ms, much of this is likely to be made up of the SQRT function. To verify this the schedule viewer, Table 4.13, offers a breakdown of latency sources. As with reading, pipelining the writes back to memory could be beneficial if a pipelined architecture were applied to the overall architecture.

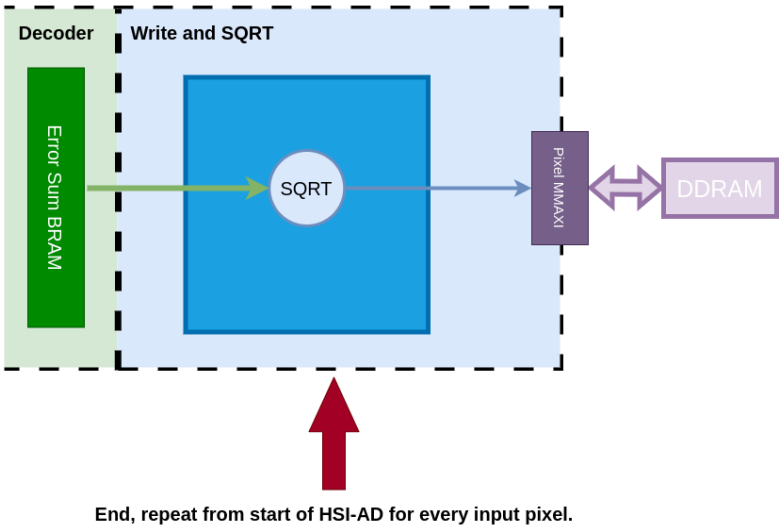


Figure 4.10: Write task initial implementation.

Write and SQRT Submodule	Loop II	Latency	Trip count	Total Latency
Write and SQRT	-	30	1	30

Table 4.12: Initial write characteristics.

Write and SQRT Submodule	Off-chip Memory	On-Chip Memory	Calculations	Control / Other	Total Latency
Write and SQRT	2	1	25	2	30

Table 4.13: Initial write schedule.

Full architecture of AE with Reconstruction Error

The overall architecture, as presented in Fig. 4.11 can be seen to be straightforward without feedback from a later execution stage. This provides a range of possibilities for improvements in terms of pipelining and as shown in the task concurrency graph there is nothing being applied yet at this stage. Possible optimisations can also be identified. Firstly memory accesses must be kept as much on-chip as possible. This means weights are to be loaded into BRAMs or other forms of memory before the design starts its calculations. Secondly, for the purposes of parallelism the design can have the different pipeline pragmas applied such that all of the design units are running at the same time. For example the dataflow pragma would create an architecture for a coarse grain pipeline without an increase of resources outside of duplicate memory shared across stages. Thirdly, unrolling of the internal loops of the RBMs is necessary for optimisations of pipelining otherwise there would be a large amount of stalls in the pipeline. Lastly, a concern would be the

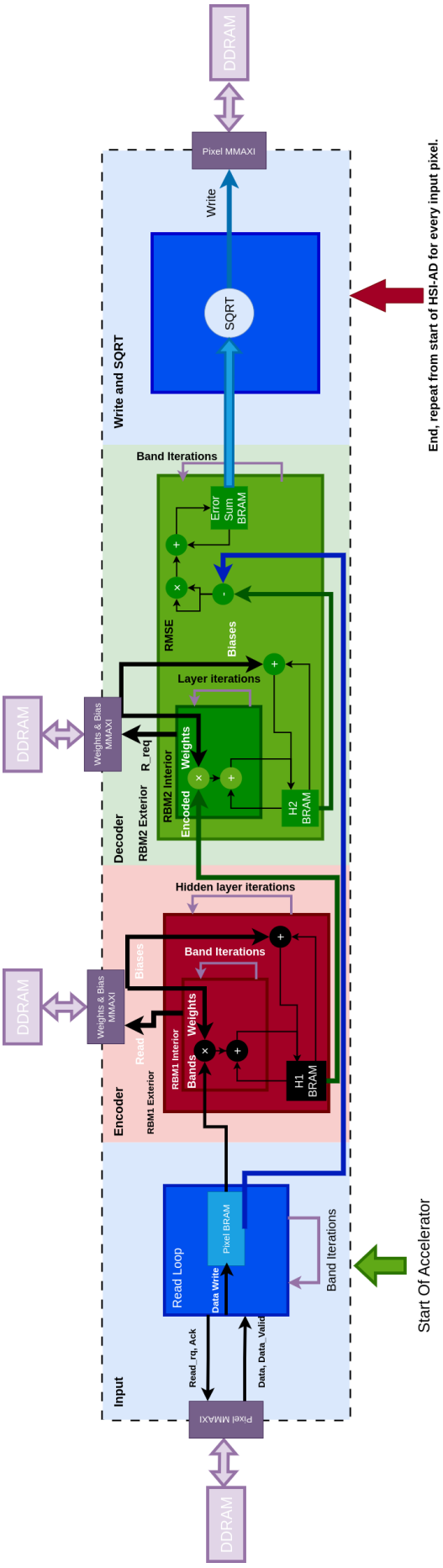


Figure 4.11: Full architecture.

multiplications and additions are implemented using DSP slices in the FPGA which could be a resource bottleneck when unrolling and pipelining take place.

4.2.2 Baseline Architecture with On-Chip Weights and Biases

Firstly the addition of BRAMs into the design without any of the other changes, see Fig. 4.11, will be measured in synthesis and co-simulation. This would come as an extra task before starting the accelerator main loop. With a memory declaration these BRAMs are written to by a loading stage. This stage reads the parameters of RBM1 from DDR4 RAM through a loop into a BRAM and once again into a separate BRAM for the parameters of RBM2, this can be seen in Fig. 4.12.

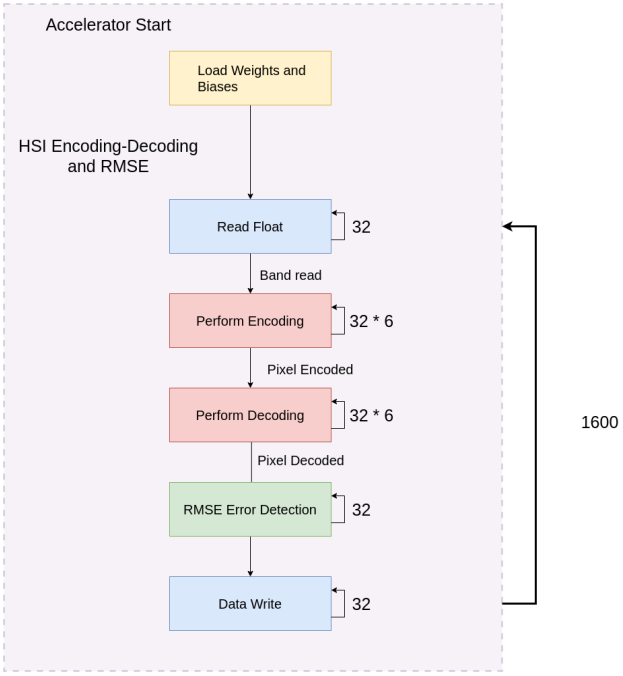


Figure 4.12: Dataflow graph with BRAM loading.

Results

Addition of loading parameters into on-chip memory

Loading the parameters adds 626 clock cycles to the design according to the co-simulation, see Table 4.14.

Parameter	Latency	Size(B)
Weights	**588**	**768**
Biases	**138**	**152**

Table 4.14: Latency impact from loading parameters.

Task	Latency (Avg)	Initiation Interval (Avg)
Encode-Decode RMSE	13466225	-
Load Parameters	626	-
Read Bands	98	8416
RBM1 Exterior	3181	8416
BRM1 Interior	481	1402
RBM2 Exterior & MSE	5121	8416
RBM2 Interior	91	262
Write and Sqrt	17	8416

Table 4.15: Latency breakdown of the sub-tasks.

As expected, the addition of BRAMs reduced the latency of the design, Table 4.15. A breakdown of where this reduction occurred can be seen in Table 4.16, the latency reduction has been highest where weights and biases were used for calculations. There have been no changes to the architecture outside of loading these weights.

Task	Original Latency	Latency BRAM	*Speedup %*
Encode-Decode RE	20860804	13466225	35%
Load Parameters	-	588	-
Read Bands	98	98	0%
RBM1 Exterior (w/o Int)	295	295	0%
RBM1 Exterior (w Int)	5485	3181	42%
BRM1 Interior	865	481	45%
RBM2 Exterior & RE (w/o Int)	2209	2209	0%
RBM2 Exterior & RE (w Int)	7425	5121	31%
RBM2 Interior	163	91	45%
Write and Sqrt	17	17	0%

Table 4.16: Comparison of latency of submodules of original and version with BRAM.

Discussion

The BRAM addition has had a performance impact of 35% over most of the operations that require parameters. So far the resource utilisation has not significantly increased. The next step for the architecture is to introduce different types of pipelining paired with unrolling. The exact factors and intervals are what will be experimented with and presented in the exploration chapter and so the "base" of these designs will be presented now. In terms of the system architecture pipelining is clearly applicable and advantageous in terms of performance. For pipelining to be optimal loops in both the encoding and decoding must be unrolled to prevent significant stalls. This pipeline can be applied in a fine-grain pipeline or coarse-grain manner as a dataflow pipeline. The advantage of a coarse-grain pipeline is that it does not implicitly require the full unrollings of RBM1 and RBM2, allowing it to have a lower resource consumption at the cost of some ability to optimise between dataflow sections. The advantage of the fine-grain pipeline is that it does have this ability to be optimised between the entirety of the design, which may make it more advantageous on the condition of all of the operations, weights and biases fitting on the FPGA at once.

4.2.3 Dataflow Architecture

Dataflow between segments of the design allow for a coarse-grain pipeline increasing the throughput without increasing the hardware utilisation significantly. In creating a dataflow design it is important that the initiation interval of every stage of the pipeline is as equal as possible to prevent long pipeline stalls, for this reason as in Fig 4.13 the RE has been separated from RBM2. To enable the design to be dataflow capable it is essential that any stage in the presented design only consumes resources from a prior stage and not a future stage. It is also necessary that data

written in any current stage is not overwriting data before it is needed in a future stage, as a WAW hazard. As the design processes concurrently it requires multiple instances of any read, encoded, or decoded band data as they are used once to be written to in their earliest stages and then to be read from in a later stage, this solved in Vitis HLS through ping-pong buffering.

Presented individually in Fig. 4.17, it is important to note that the RE module is its own distinct CU. This is an optimisation to decrease the coarseness of the pipeline and improve the latency by allowing for more parallelism. The corresponding table 4.17 shows the dataflow characteristics with and without submodule pipelining. This serves as an introduction to directives that will be applied over the design exploration. When a loop is pipelined any loops below it in its hierarchy are implicitly unrolled. This also generates a need for a higher memory bandwidth. For this reason memory partitioning is performed to instantiate more memory objects to take advantage of the increase in read/write ports to achieve the required bandwidth. This partitioning can be done manually or automatically in Vitis-HLS.

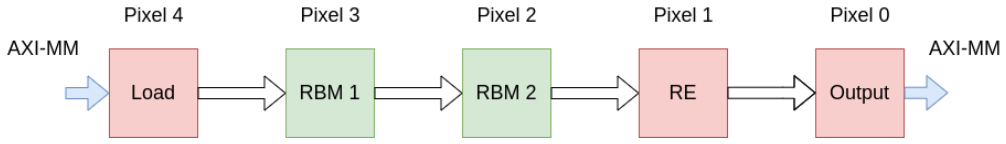


Figure 4.13: Coarse grain pipeline processing pixels in parallel.

Task (Avg)	**BRAM (Baseline)**	**Dataflow**	**Speedup %**	**Dataflow & Pragmas**	***Speedup %***
Encode-Decode RMSE	13466225	7175885	47%	507540	96.2%
Load Parameters	588	577	2%	577	-
Read Bands	98	104	-1%	128	-131%
RBM1 Exterior (w/o Interior)	295	295	0%	-	-
RBM1 Exterior (w Interior)	3181	3181	0%	316	90%
BRM1 Interior	481	481	0%	Unrolled	-
RBM2 Exterior (w/o Interior)	2209	1632	27%	-	-
RBM2 Exterior (w Interior)	5121	4481	19%	134	97%
RBM2 Interior	91	91	0%	Unrolled	-
Write and RE	17	768	-451%	293	-172%

Table 4.17: Dataflow design comparison with and without directives.

Discussion

The initiation interval of a coarse-grain pipeline is the highest latency of the different pipelined tasks. This results in an initiation interval of 316 between pixels, as seen in Table 4.17. This is caused by the RBM1 task because it is the highest latency CU in the coarse grain pipleine. The organisation and internal pipelining of these CUs can be visualised in Fig. 4.14, where the depth of the pipeline is the number of code sizes in RBM1 and the number of bands in RBM2. The stages involved in each loop are based on accommodating the parallel processing of each of these output nodes of the respective RBMs. Looking further at the graph there is a time between the decoding task being completed and the encoding task still executing where there is no utilisation for decoding. The same is true for RE. This is not the most efficient implementation possible in actual utilisation of FPGA allocated resources. To achieve a lower initiation interval and higher utilisation of allocated resources a more fine-grained approach would be beneficial.

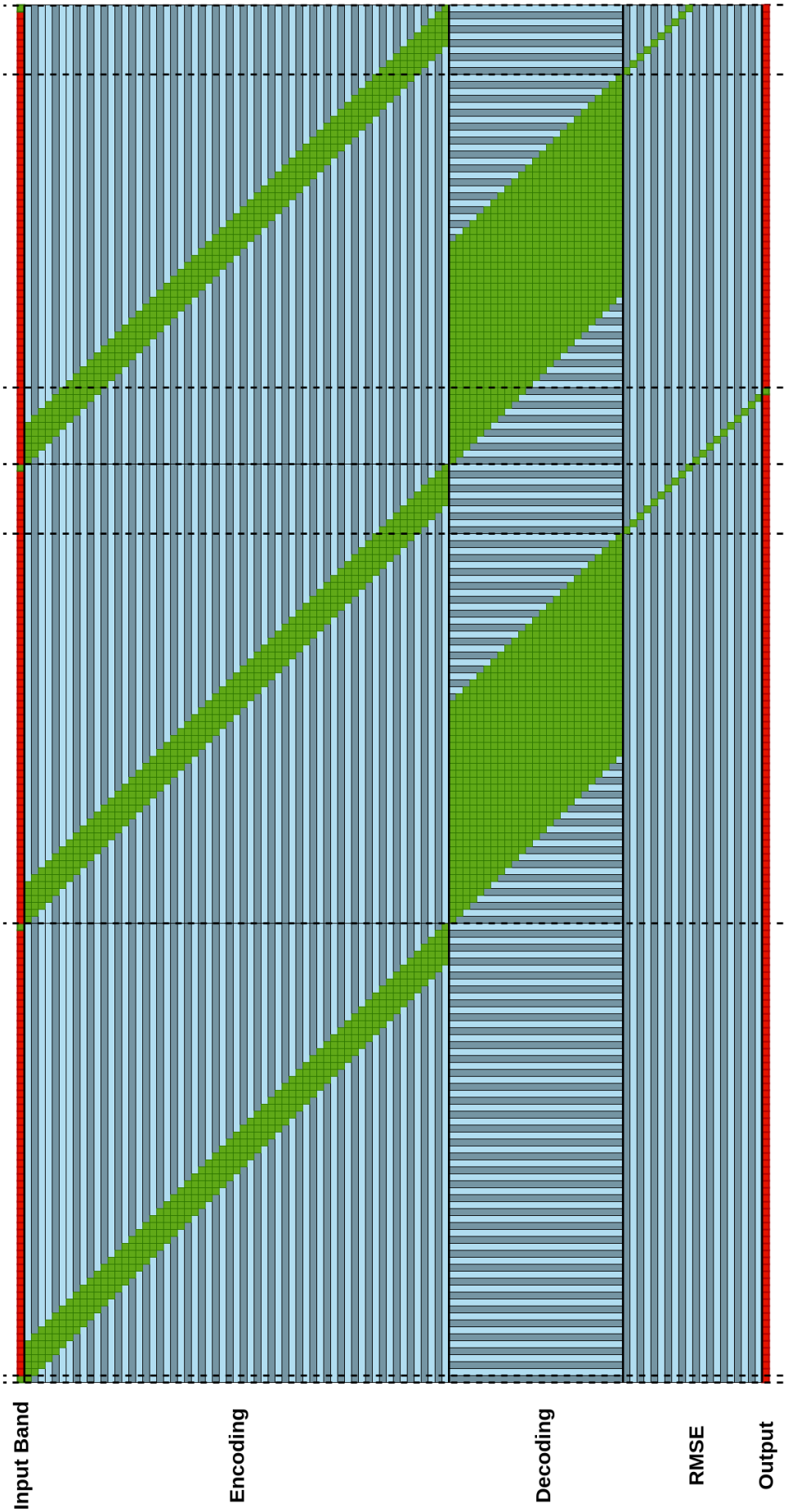


Figure 4.14: Coarse-grain pipeline with internal fine-grain pipelines of each task.

4.2.4 Fine-Grain Pipelined Design

To achieve a fine-grain pipeline a pipeline directive is applied for both the encode and decode tasks. This means RBM1 and RBM2 have to be fully unrolled, potentially requiring significant resources in the form of DSPs, memory and LUTs. Although large amounts of resources are required there are benefits to this approach, as having this pipeline covering both RBMs means they are both in each others scopes in terms of automatic HLS optimisations. This automatic inlining means HLS will optimise between the unrolled loop boundaries. This results in flexibility in terms of setting an appropriate initiation interval that will ensure the entirety of the design is optimised for that specific interval. If the highest throughput, initiation interval 1, has too high a utilisation, increasing this interval will reduce resources drastically. Another issue with a low II pipeline is the fact that to accommodate for this, all on-chip memory is mapped to LUTs in the design rather than BRAMs or URAMs. This is because the bandwidth requirement becomes very aggressive and URAMs or BRAMs are a form of higher bandwidth low density memory.

Task (Avg)	**BRAM (Baseline)**	**Dataflow & Pragmas**	***Speedup %***	**Pipeline II = 32**	***Speedup %***
Encode-Decode RE	13466225	507540	96.2%	52295	99.6%
Load Parameters	588	577	-	484	18%
Read Bands	98	128	-131%	Pipelined, 34	65%
RBM1 Exterior (w/o Interior)	295	-	-	Unrolled	-
RBM1 Exterior (w Interior)	3181	316	90%	Unrolled	-
BRM1 Interior	481	Unrolled	-	Unrolled	-
RBM2 Exterior (w/o Interior)	2209	Unrolled	-	Unrolled	-
RBM2 Exterior (w Interior)	5121	134	97%	Unrolled	-
RBM2 Interior	91	Unrolled	-	Unrolled + (RE)	-
Write and RE	17	293	-172%	Pipelined (No RE)	82%

Table 4.18: Comparison of latency of pipelined, dataflow with directives and original BRAM accelerators.

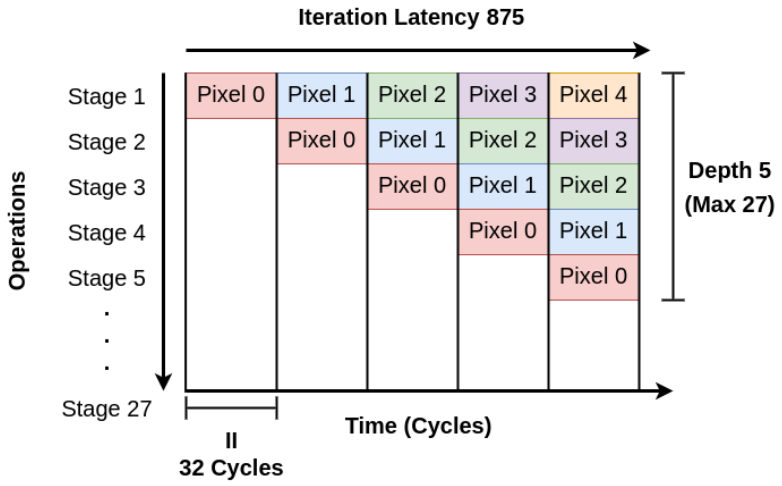


Figure 4.15: Fine grain pipeline processing pixels in parallel.

Discussion

As can be seen in Table 4.18 this initial fine-grain pipeline has an advantage in terms of throughput. A part of this advantage comes from the unrolling that is performed over the entirety of RBM1 and RBM2 which creates an iteration latency of 875 as seen in Fig. 4.15. This lower latency alongside the division of these into 27 pipeline stages means up to 27 pixels are processing in the pipeline at once. Outside of this pipeline the reading and writing of pixels is also in a coarse-grain dataflow pipeline and performed in parallel. With the II of 32 the design is actually not consuming too many resources. The most used resources are LUTs and FFs indicating that partitioning is applied due to the higher bandwidth needed. There are still opportunities for a faster implementation for a fixed parameter PCA in the exploration, as the II can still be lowered.

4.2.5 Reordered Design

The current pipeline and dataflow implementations are organised such as in Fig. 4.16. The position of the first executed synapses are indicated by the bold lines. This means that in this loop ordering all of the input bands are multiplied by their weights and accumulated for each code sequentially without contributing to codes in an interleaved manner. This is problematic as it requires that every band is loaded on-chip before it is processed, creating a dependency in the input data and the processing for each code in RBM1. An alternative ordering which allows for the opportunity for input band tiling is to swap the loops, such that a single input band contributes to every code at once. This removes the input dependency and allows for a pipeline upon band entry into the NN.

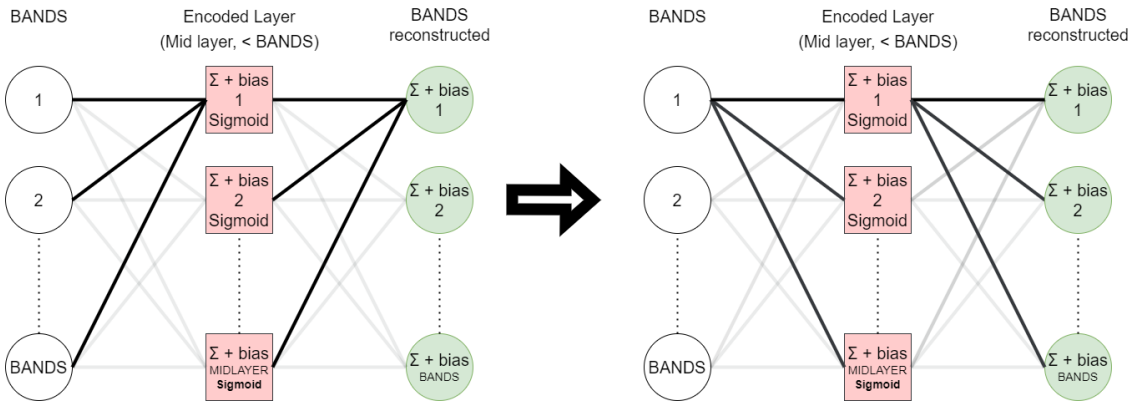


Figure 4.16: Reorder transformation on AE.

When fully pipelined as previously discussed this design becomes equivalent to the pipelined due to the implicit unrolling of the loops. As a result this will be another dataflow design to compare in the design space exploration. There are options for full unrolling and pipelining over dataflow sections and this will be explored for the following band tiled design.

4.2.6 Band Tiled Partial Reordered Design

This design aims to exploit the reordering of the input of RBM1 to allow for a tiling in the pipelining of its MAC operations. Tiling would mean that instead of reading the entire image before processing, these inputs can be read a single or multiple bands at a time and processed. This can also be applied to the RE calculation. This can be combined with the original sequential band access pattern for RBM2, see Fig. 4.17. With the benefits established of reordering RBM1, RBM2 benefits from a synchronisation in the resource access pattern if used in an pipelined dataflow configuration. This results in the pipeline depths being similar bringing their latencies closer. Accessing all codes in parallel for MACs involves less concurrent resources than accessing every band at once for the MAC of a single code. Additionally, if this were fully unrolled either loop orderings would be equivalent in the generated RTL. RBM1 is tiled by having a loop based on the AXI-MM input size covering the MAC loop visualised in Fig. 4.18, as such pipelining in this module can now be performed on an unrolled sub-partition of the reordered loop.

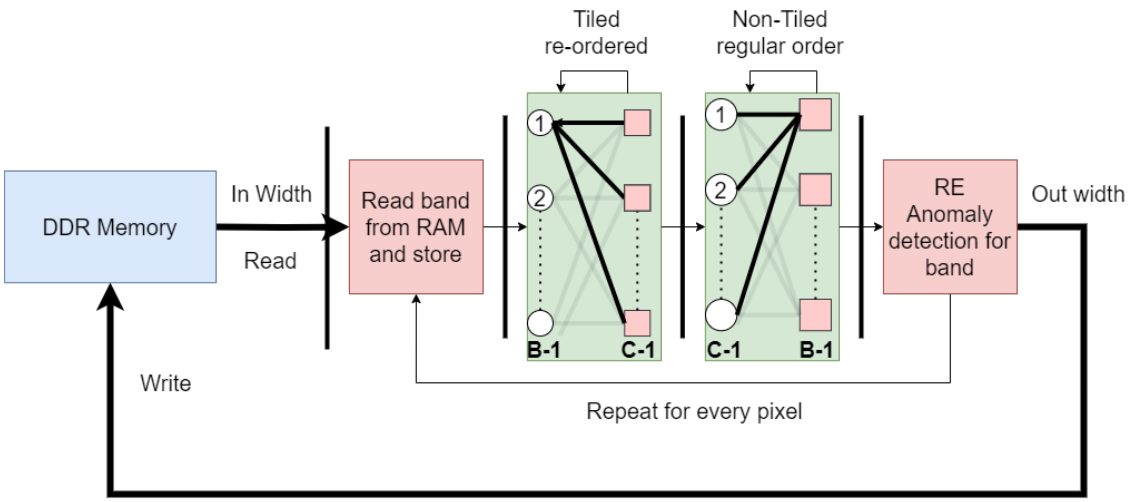


Figure 4.17: Reorder transformation on AE.

This is expected to be more beneficial at smaller input word sizes because it means less visible layer nodes would be encompassed by the tiles at a time. This also means it is even more beneficial for the case of designs with large bands.

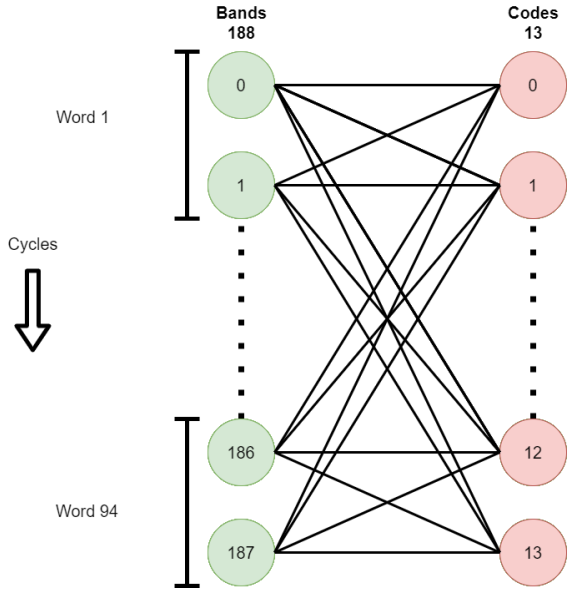


Figure 4.18: Tiling transformation on RBM1 with 64-bit word sizes.

4.2.7 Overview and Parameters of Exploration

A design space exploration will take place by varying the design parameters and also by varying the kind of images the design can accommodate.

Interface Widening

The maximum supported width of AXI-MM interfaces in Vitis HLS is 1024 bits, this is up to 32-bands. For HSIs larger than 32-bands some accommodations will have to be made. It is unclear what AXI interface width will be the best for the design on the FPGA as simulation can be misleading, in particular regarding the AXI latency. This is because this DDR4RAM is accessed by a hard bus on the SoC and the speed can depend on the bus and RAM frequency, the burst size and width of interfaces. Having a design that is too fast for the AXI bandwidth would lead to wastage in resource use.

Pipelining and Dataflow

The pipelining initiation interval can be changed, this will be experimented with both in the pipelined architectures and within the submodules of the dataflow architectures. Dataflow and coarse-grain pipelines will also be applied inside and outside of the designs, if applied surrounding a pipelined design it could mean that there is concurrency in reading and writing to memory. If placed directly outside of the pipeline it may cause the pipeline to assume data can be ready at any time which is worse for optimisation than considering the predicted AXI latency.

Unrolling

Full unrolling is implicit to any loop within a pipelined hierarchy. When designs are pipelined manually the pipeline interval to be customised and allows for an unrolled design to be executed over more cycles.

Using BRAMs, URAMs

BRAMs and URAMs are useful for lowering the utilisation of resources like FFs and LUTs. These provide a denser memory and utility. In very high throughput designs it may not be possible to apply these resources as they are a low bandwidth memory object.

Memory partitioning

Often memory resources need to be divided and the storage of data spread across them to take advantage of the increased number of memory ports, which translates to a higher peak bandwidth. Caution must be taken with memory partitioning as it is easy to manually partition for a higher bandwidth than is actually needed in the accelerator. Pipelines allow for automatic partitioning to maximise throughput without partitioning more than necessary. Dataflow requires a more manual approach which could be cause for error.

Flexibility in height, width, code side and band size

How these directives interact with the design so far have been discussed for fixed height, width, code size and bands. The ideal goal would be an accelerator that has some flexibility without degrading performance through this flexibility. The band size must be flexible to accommodate PCA and non-PCA designs in the same accelerator. This includes flexibility in the number of bands of the input image and the number of bands in the hidden code layer. Images may also vary in height and width which is an additional change to accommodate.

4.3 Double Sliding Window Implementation

The DSW for the purpose of anomaly detection was implemented into RTL using Vitis-HLS. The implementation was run with both sets of results from the autoencoder Matlab and C code being used as the input. The implementation was also tested with the C trained model. The requirements for the fixed parameters of the DSW were discussed in the golden model methodology section, however the design can be synthesised for any combination of window parameters and thus this will be the main exploration parameter for a hardware implementation. It was not possible to simply copy and paste the HLS C architecture directly into the implementation. The adaptations required, and the reasons behind them will be discussed in the C to HLS section. The architecture section will go over what the intended implementation architecture is and the different HLS exploration directives that can be applied to the design. This will then be experimented with in the exploration section.

4.3.1 From C to HLS

Step 0 : Padding

The high-level C code was developed with the intention of not processing out of bound pixels by utilising 'if' conditions to check if the window positions are out of the image boundary. This

methodology is acceptable for a processor as it is able to perform branching with prediction and by eliminating out of bound data it is possible to save time processing. Dedicated hardware differs from this, as resources would have to be allocated on the chance the condition 'the pixel is within bounds' is true. This generates extra logic for checking the condition is met, after already processing the result. This creates complex control logic layered around the hardware resources that would be allocated based on the condition. This control logic for checking every pixel index for every window position for every feature matrix would quickly become burdensome for the HLS to schedule and allocate. Instead for HLS, impossible data is placed into a buffer region. The choice of this data is based on reconstruction errors only being a value between 0 and 1 and always positive. -1 is placed into the on-chip buffer before reading image data as can be seen in Fig. 4.19.

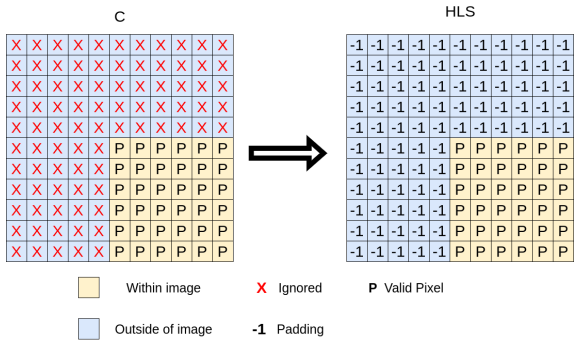


Figure 4.19: Out of bound strategy using padding in HLS.

Before observing the HLS code that performs the key calculations of the DSW, it is necessary to understand the utilised pre-processing variables in Table 4.19. These are calculated at compile time for the DSW, and as such there is an avoided utilisation overhead that would come from having to calculate these variables during the running of the DSW as an Intellectual Property (IP) core.

Pre-processing Variable	Calculation	Value	Comment
WIN_SIZE	1	1	Inner Window Size
WIN_DIF	5	5	Spatial distance from edge of inner to outer window
WIN_LEN	2*WIN_DIF + WIN_SIZE	11	Length of single side of entire window
HALFWAY	WIN_DIF + (WIN_SIZE-1)/2	5	Index of Pixed Under Test (PUT)
BOUNDARY_TL	WIN_DIF	5	Indicates left and top boundaries of inner window
BOUNDARY_BR	WIN_LEN - WIN_DIF	6	Indicates bottom and right boundaries of inner window
NEIGHBOURS	WIN_LEN^2 - WIN_SIZE^2	120	Surrounding pixels used in RE and code calculations
MAX_CODE	16	16	Maximum supported code size per pixel

Table 4.19: Pre-processing variables used in HLS C code for window (1,5).

Step 1 : Mean

The idea is that the center pixel of the window can be compared to -1 and if they are equal it is not a legitimate data pixel and thus the result of the sliding window for this pixel can be discarded. This is important because padding data needs to be inserted before the image and between image lines, to act as the right side padding for the current row and the left side padding for the next row of image data. It can then become difficult to distinguish padding from real data. The choice of -1 means that when the summation happens to calculate the neighbourhood mean of reconstruction errors, the values that are not -1 can be counted. For any fixed window size the number of pixels in a neighbourhood can be calculated at compile time. The number of neighbourhood pixels subtracted by the number of legitimate pixels can be added to the total sum to offset the number of non-legitimate -1 pixels. This removes the need for any boundary checking in the mean and code to perform this is illustrated in Listing 4.1.

Listing 4.1: HLS code for RE mean

```
total_loop:
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
```

```
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
            r_total = r_total + re_mat[y][x];
            if(re_mat[y][x] != -1) {
                count++;
           }
        } else{
            continue;
        }
    }
}
r_total = r_total + (NEIGHBOURS - count);
r_mean = r_total/count;
```

Step 2 : Standard Deviation

Once the mean is counted the standard deviation of neighbour reconstruction errors in the window can be calculated. Before this calculation the -1 pixels are replaced by the mean, so that they do not impact the calculation. As the standard deviation uses root average squared difference from the mean these padded values are cancelled out. This average is the average of legitimate pixels, by using the count of non-padding values from step 1. How this is achieved is shown in Listing 4.2.

Listing 4.2: HLS code for standard deviation

```
pad_mean:
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
            if(re_mat[y][x] == -1) {
                work_mat[y][x] = r_mean;
            } else{
             work_mat[y][x] = re_mat[y][x];
            }
        } else {
            continue;
        }
    }
}
square_difs:
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
             dat_t comp;
             comp = work_mat[y][x] - r_mean;
             std_dev = std_dev + (comp * comp);
        } else {
             continue;
        }
   }
}
std_dev = sqrt(std_dev/count);
```

Step 3 : Weightings

With the standard deviation calculated, every RE neighbour pixel is checked to see if it deviates from the mean. If it deviates by beyond a standard deviation its corresponding code weighting is penalised to 0, as specified in the penalty factor. If it does not deviate beyond the standard deviation then it is replaced by its RE inverse multiplied by the PUT as specified in Novel-AWDBN. All non legitimate (padding) pixels will always deviate from the mean by above a standard deviation, and as a result they will have a penalty factor applied. This removes the need to consider a padding value for their code distance, which will just be padded with 0s synchronously with RE padding.

Listing 4.3: HLS code for weighting and penalty

```
weight_pens:
```

```
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
            if(abs(re_mat[y][x] - r_mean) > std_dev ) {
                weight[y][x] = 0;
            } else {
                weight[y][x] = re_mat[HALFWAY][HALFWAY]/re_mat[y][x];
            }
        } else {
        continue;
        }
    }
}
```

Step 4 : Code Distance

With the weights calculated, the code distance from the code of the PUT is calculated for every neighbourhood pixel. This is done as specified in the algorithm, with the root of squared distances and can be seen in Listing 4.4. This is the most problematic part of the algorithm for mapping to the FPGA as performing this step for floating point is very resource intensive and difficult to route for large window sizes.

Listing 4.4: HLS code for neighbour code distance

```
zero_dist:
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
            code_dist[y][x] = 0;
        } else {
            continue;
        }
    }
}
code_distance:
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
            for(int c=0; c < MAX_CODE; c++) {
             if( c < codes) {
                dat_t temp;
                temp = code_mat[y][x][c] - code_mat[HALFWAY][HALFWAY][c];
                code_dist[y][x] += temp * temp;
             } else {
                break;
             }
           }
            code_dist[y][x] = sqrt(code_dist[y][x]);
        } else {
            continue;
        }
    }
}
```

Step 5 : Anomaly score

The anomaly score of the PUT is then calculated as the sum of these euclidean distances times their respective weighting. Based on whether or not the pixel is legitimate or not this score will then be written or discarded. Buffer filled is not a test of the entire buffer, as the buffer is pre-filled in a separate low latency loop, rather it is a test of whether or not the buffer values have propagated the window matrix as this is where the -1 test occurs and may have old values from the previous run of the accelerator. If the buffer has been prefilled with -1s and a -1 is not currently the PUT, this is demonstrated in Listing 4.5.

Listing 4.5: HLS code for anomaly score calculation and verificaton

```
anomaly_score:
for(int y = 0; y<WIN_LEN; y++) {
    for(int x = 0; x<WIN_LEN; x++){
        if ( !(((x >= BOUNDARY_TL) && (x < BOUNDARY_BR))
        && ((y >= BOUNDARY_TL) && (y < BOUNDARY_BR)))) {
            anomaly_score += code_dist[y][x] * weight[y][x];
        } else {
        continue;
        }
    }
}
// Send score
if((value != -1) and buf_prefill == 1) {
    score_stream.write(anomaly_score/count);
}
```

Concerns and mitigation strategies

There are a few concerns regarding the optimisation of this buffering strategy, firstly it is key that the buffer is filled and the first pixel is centered in the window before processing begins to save latency in the design and improve synchronisation with the autoencoder component. If the buffer were to be filled by the processing loop, which would have a large initiation interval, this would mean that buffer data is only placed every initiation that includes the critical path of the design. Rather than this, data can be placed in before the main loop is initiated. The intention for highest throughput for a resource threshold is that the AE and the DSW have a matching loop initiation interval, as the output for a single AE processing loop is the input for a single DSW processing loop, their synchronisation is key for a dataflow architecture. Another issue is that there is a dependency in the shifting matrix only being shifted after it is already finished being used in processing. These operations can be interleaved by copying or buffering the matrix rather than using it for processing directly. This should be automatically optimised for any coarse-grain or fine-grain pipelined system in Vitis-HLS.

4.3.2 Architecture

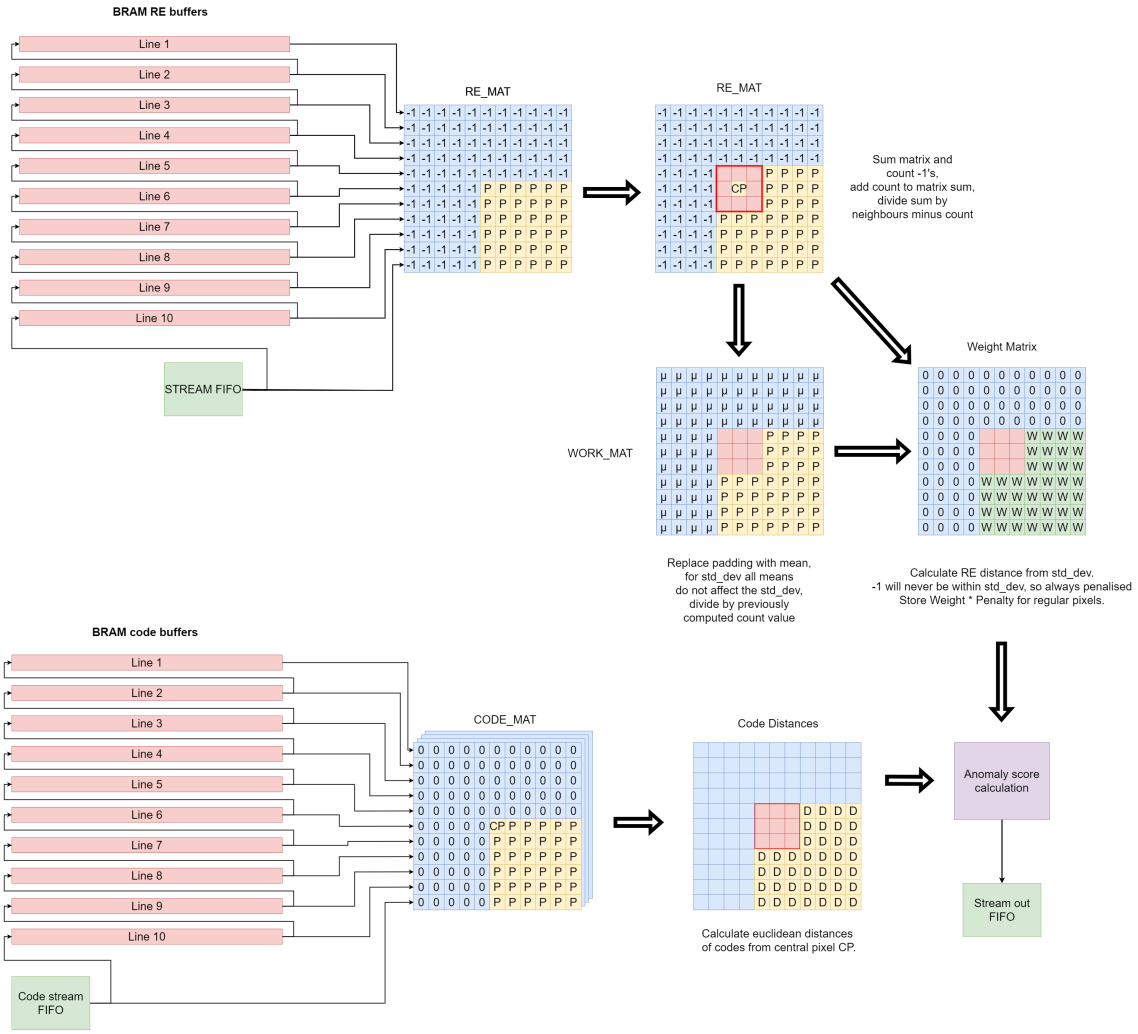


Figure 4.20: High level architecture of AWDBN double sliding window implementation for Win(3,4).

Interfaces

There is a flexibility in the choice of architecture. The DSW and AE can be integrated as a single component or kept as separate standalone components. In the case of an integrated component there would need to be an AXI-Stream connection of code and reconstruction errors of every pixel fed from the AE into the DSW, with the AE having an AXI-MM for reading data and the DSW an AXI-MM for writing data. If standalone both components need to contain AXI-MM for reading and writing data.

Shifting Line Buffer

To implement the sliding window for the RE and codes of each pixel, a shifting line buffer is used and can be seen in 4.20. This line buffer holds all the rows of image data for the image rows the sliding window is operating on. As data is sliding through the buffer it is replaced with the data of the next line. This is performed by each row of the line buffer writing into the row above, with the bottom being fed data from the AXI-MM or stream. This is effectively shifting left and upwards. As only one shift is required every time a pixel is processed this only requires a low bandwidth, so BRAMs become the natural medium for implementation. There is one BRAMs for each line to take advantage of the read and write ports so that only the exact amount of required bandwidth is needed. The shifting in these BRAMs is not physical shifting in hardware but rather controlled

by a incrementive pointer through this memory. When the pointer reaches the end of a data line it returns to the beginning, overwriting old data with new incoming data. Each pixel calculation initiation the data input into the BRAMs becomes the input to a shift matrix which performs the sliding window operation. This matrix physically shifts every clock cycle without a pointer.

Window Matrix

With the shifting line buffer acting as a low bandwidth line cache, the shift-matrix is a high-bandwidth line buffer fed processing cache that is intended for unrolling and pipelining on the FPGA to be used in calculations. This is implemented in LUTs and FFs for high bandwidth partitioning. There is a shift matrix for both the codes of each pixel contained and the reconstruction errors. The processing of these is exactly as described in C to HLS, with intermediary matrices, including; Mean padded working matrix, Weight Matrix, and Code distance matrix. These are the size of the entire window and data is shifted from the right-to-left one column at a time as the window slides along an image. There are two matrices, one for reconstruction errors, one for codes, this can also be seen in 4.20. The code buffer is significantly larger than the reconstruction error buffer as it contains all of the code bands as a 3rd dimension. The intention of these matrices is to implement the double sliding window and that, with fixed internal and external window parameters, the center of this window will not contribute to scheduling, calculations or need partitioning. It is not used in any calculations, only for shifting.

Cache for calculations

The calculations will require a large latency over many calculations. To avoid complex routing from a single memory array, the different phases of the calculations are split among separate matrices as is notable in 4.20. The only sliding matrices are the code window and RE window. There are more for temporary storage. Firstly the mean is calculated from the sliding window, through the C to HLS highlighted method. With the padding replaced by the mean, the entirety is placed into a fast register-based "working matrix" which is also a collection of FFs and LUTs. Once this is written to, it is used in the standard deviation calculation which is then stored into a register. Now the RE matrix is once again used to calculate, if a neighbour RE at index i,j varies from the mean by a standard deviation, a weight matrix has 0 stored at i,j, otherwise its inverse multiplied by the PUT is stored. The RE matrix is not needed in calculations after this point but it is intended to remain synchronous with the code matrix which still needs to be used for a calculation. It cannot be shifted and the next loop iteration cannot yet begin. At this point the code distance is calculated from the code window matrix by storing into temporary registers the bandwise square difference between the PUT and all the neighbourhood pixels. This is summed and square rooted into a code difference matrix, which is used for calculating the anomaly scores.

Output stream

Finally the anomaly score is calculated as the summation of each element of this difference matrix cache consisting of the neighbour code difference with their corresponding RE weighting and this is written into a single register. Finally this is written into a stream FIFO, to be output by the AXI-MM to memory.

4.3.3 Testing

Continuous testing was performed during the development of the accelerator through comparison with the golden reference C code. This was performed for C simulations of the hardware accelerator and co-simulations of the hardware accelerator. The C simulation was used for the purposes of validating the result from the accelerator, while the co-simulation is used in benchmarking and optimisation alongside RTL validation. Running a co-simulation can allow the optimal FIFO sizes to be found, however these sizings vary based on exploration parameters. Finding the most locally optimal should be left until the selection of a final design as it is too high effort to be done through trial and error.

4.3.4 Double-Sliding Window Exploration Parameters

The adjustment of window parameters will be key for finding a balance between speed and accuracy. Changing the internal and surrounding window sizing can significantly reduce the amount of floating point operations and thus, the schedulability and resource utilisation of the hardware accelerator on the FPGA. An example is given for Window(5,3), Fig. 4.21 with an inner window size of 5 and outer window difference of 3. This creates a 11x11 window and the number of calculation neighbours varies for these parameters. The design is synthesisable for any combination of parameters, which are determined by pre-processing variables in the HLS.

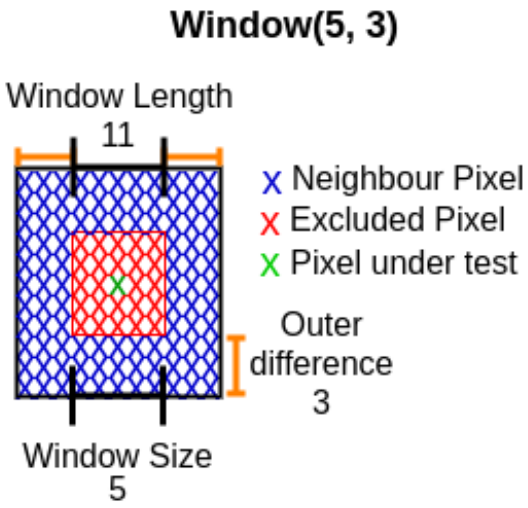


Figure 4.21: Parameters of DSW.

Dataflow and Pipelining

As performed in the autoencoder implementation segment, there are some helpful HLS directives that can be applied to the design in the pursuit of finding a suitable speed and resource tradeoff. For a large window pipelining will most likely not be possible unless the initation interval is very large, and in this case having a coarse-grain pipeline would be preferable.

Code size

The supported code sizes is another important factor, in Gunderson's thesis the code size of 13 was seen as optimal in terms of processing and performance for the Matlab trained NN, but for the C trained, after some experimentation this has been found to differ slightly, so the accelerator will have to accommodate a range of code sizes. The maximum code size has the largest implication on performance.

Max Image Width

The max supported image width has implications on the BRAM size that can be supported for the image. For images with a large width storing a whole line in memory may become prohibitive. As this parameter is most likely to be independent of the other directives and factors that can be applied this will not need to be explored significantly beyond establishing the whether the maximum supported image size of 1024 by 1024 can be accommodated in this design.

Integration

Options for integrating the DSW and the autoencoder include having a stream between the designs as separate accelerators, fully integrated into a single accelerator or both individually communicating with memory directly. This will be looked at as part of the exploration.

Exploration windows

For combination with AEs in the design space exploration, a few candidate windows have been synthesised for different window configurations. A key note is that only Window(1,5) has a reduction in size when synthesised for a higher II and lower speed. The other windows seem to increase in resource utilisation. This may be because they require some kind of synchronisation padding using registers and LUTs. For this reason, for all windows except Win(1,5) Table. 4.21 windows will be used with all AEs and for AEs with an II of 240 or above 4.20 will be used.

Width	Height	Code	WIN	DIF	Initation Interval	BRAM %	DSP %	FF %	LUT %	URAM %
150	150	16	1	5	224	27	4	72	61	0
150	150	16	1	4	224	21	3	42	50	0
150	150	16	3	3	224	21	3	39	47	0
150	150	16	5	2	224	21	2	33	39	0
150	150	16	7	1	224	21	1	24	28	0

Table 4.20: Lower speed window properties.

Width	Height	Code	WIN	DIF	Initiation Interval	BRAM %	DSP %	FF %	LUT %	URAM %
150	150	16	1	5	120	27	8	72	80	0
150	150	16	1	4	120	21	5	49	56	0
150	150	16	3	3	120	21	5	47	52	0
150	150	16	5	2	120	21	4	39	44	0
150	150	16	7	1	120	21	2	29	32	0

Table 4.21: Higher speed window properties.

Chapter 5

Results

This chapter presents the results of the project. The first section describes results from the exploration of the AE hardware architectures suitable for combination with DSWs. The second section provides results of the synthesised estimates of the full algorithm when combining various AEs with the DSWs. Finally the results will be summarised in relation to the original aims of the project through the identification of physically possible combinations for a flexible adaptation.

5.1 Exploration of Hyperspectral Anomaly Detection

This section describes a design space exploration which was carried out to investigate the possible hardware architectures that could be used for the implementation of the AE designs. Also described are two modular DSW implementations that could possibly be connected to these AEs through a streaming interface. When assessing which of the options to choose consideration will be given to the potential for tradeoffs in accuracy and resource utilisation alongside throughput.

5.1.1 Exploration of Fixed Band and Code Size AEs

This section presents an exploration of applying AEs to the ABU dataset images, specifically analysed for height, width 100x100. The AE designs are able to accommodate any height and width of image as the processing is performed a pixel at a time. The designs that will be explored have fixed band and code sizes. They have been unrolled, pipelined or converted into a dataflow design by the usage of directives in Vitis-HLS. The degree of memory partitioning and memory object implementation has been left to the tool to decide where possible as it has not been surpassed through manual input. A breakdown of the Computation Unit organisation for each architecture is presented in Table 5.1. Full details of the results are included in Appendix A, which provides the parameters, unrolling, pipelining and intervals used in the generation of the designs beyond the base architectures. Cases where the partitioning factor has been set manually are available in the Appendix A.

	Computation Unit						
Architecture	RMB1	RMB1-I	RBM1-O	RBM2	RBM2-I	RBM2-O	RE
	External Loop Of Encoder	Internal Loop Of Encoder	Sigmoid Of Encoder	External Loop of Decoder	Internal Loop of Decoder	Sigmoid Of Decoder	Reconstruction Error
Dataflow	Y	In RBM1	In RBM1	Y	In RBM2	In RBM2	Y
Pipelined	Y	In RBM1	In RBM1	Y	In RBM2	In RBM2	In RBM2
Reordered	Y	In RBM1	Y	Y	In RBM2	Y	Y
Retiled	Y	In RBM1	Y	Y	In RBM2	In RBM2	Y
Y indicates that there is a distinct CU, otherwise integrated into the CU indicated.							

Table 5.1: Table of distinct CU units that apply for each architecture.

Graphs of utilisation vs execution time

The following graphs show the results of the exploration. They display different AE candidate architectures with different directives applied. This is organised as -

- Utilisation Vs Resources for 19 Band, 6 Code layer size AEs is presented in Fig. 5.1 with corresponding Appendix Table A.1.
- Utilisation Vs Resources for 32 Band, 8 Code layer size AEs is shown in Fig. 5.2 with corresponding Appendix Table A.2.
- Utilisation Vs Resources for 120 Band, 13 Code layer size AEs can be viewed in Fig. 5.3 with corresponding Appendix Table A.3.
- Utilisation Vs Resources for 188 Band, 13 Code layer size AEs is displayed in Fig. 5.4 with corresponding Appendix Table A.4.

All designs are from modifications to the base architecture with on-chip memory in Fig.4.11. There is also an additional code output stream that outputs the codes for every pixel processed. Dataflow corresponds to a coarse-grain pipelined derivative such as Fig. 4.13. Pipelined to Fig.4.15, Reorded is a coarse-grain permutation as in Fig.4.16, and Retiled encompasses pipeline and dataflow permutations of the partial-reorded tiled architecture in Fig.4.17.

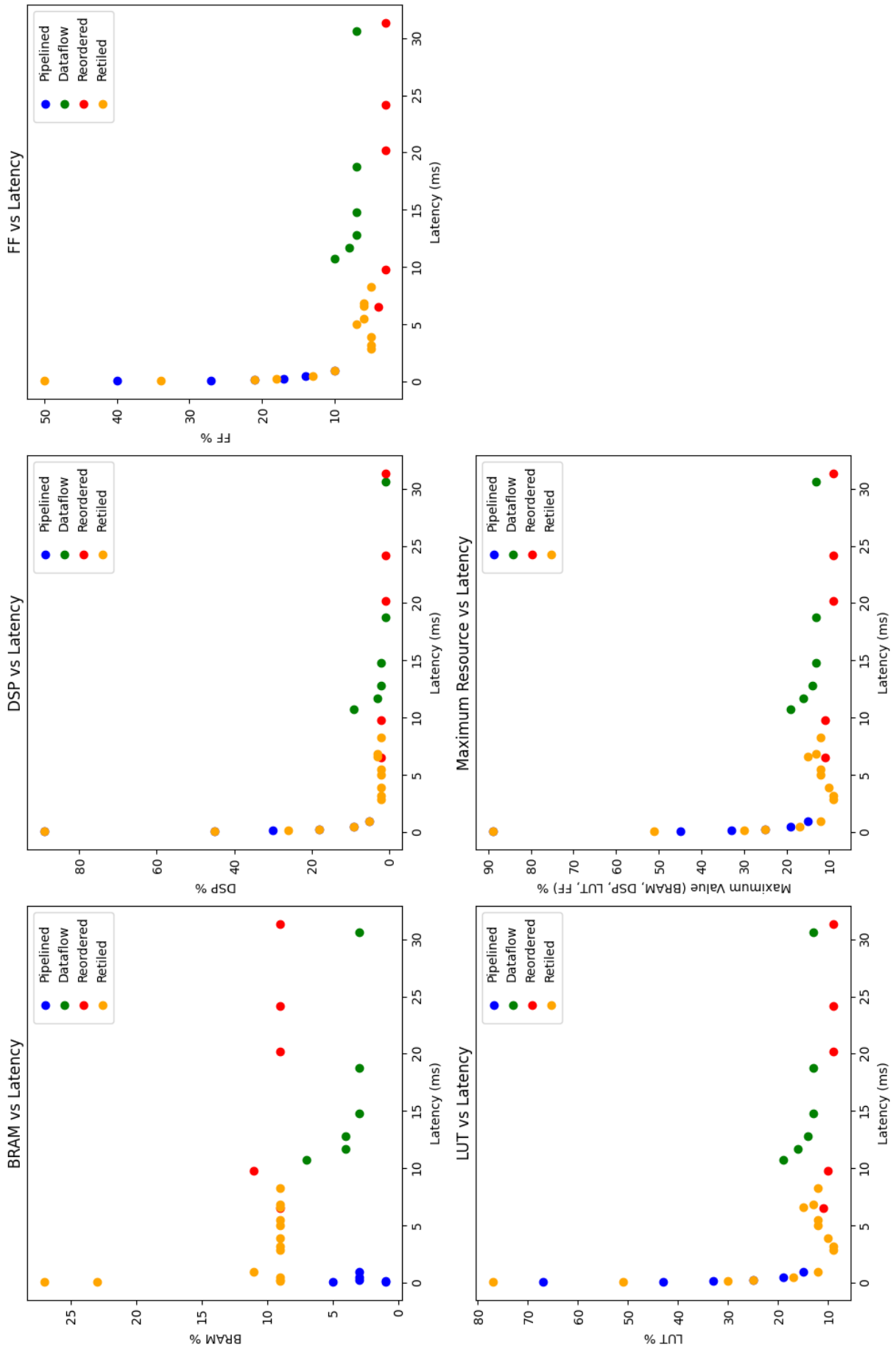


Figure 5.1: Utilisation vs latency of 19 band 6 code autoencoders.

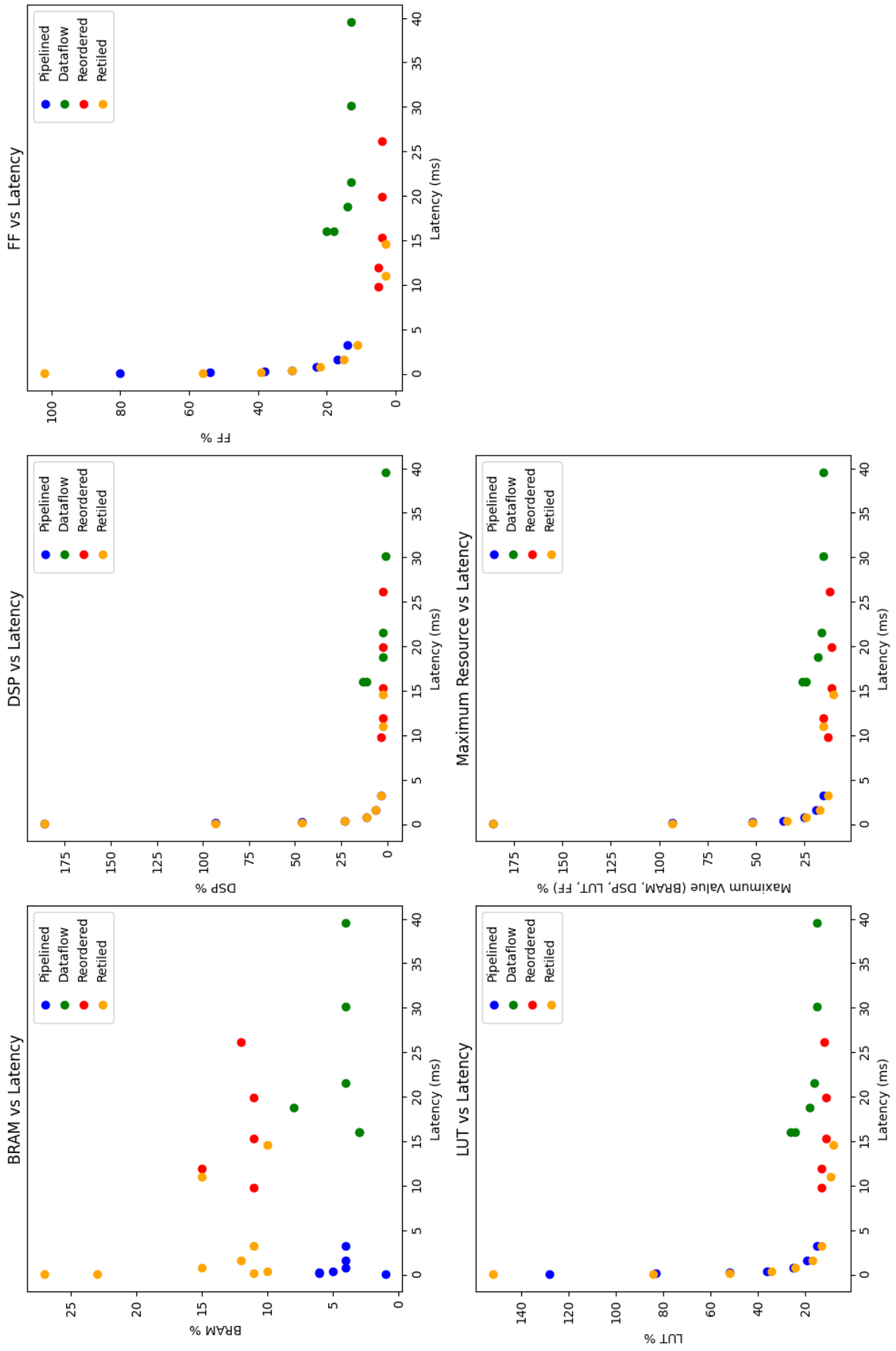


Figure 5.2: Utilisation vs latency of 32 band 8 code autoencoders.

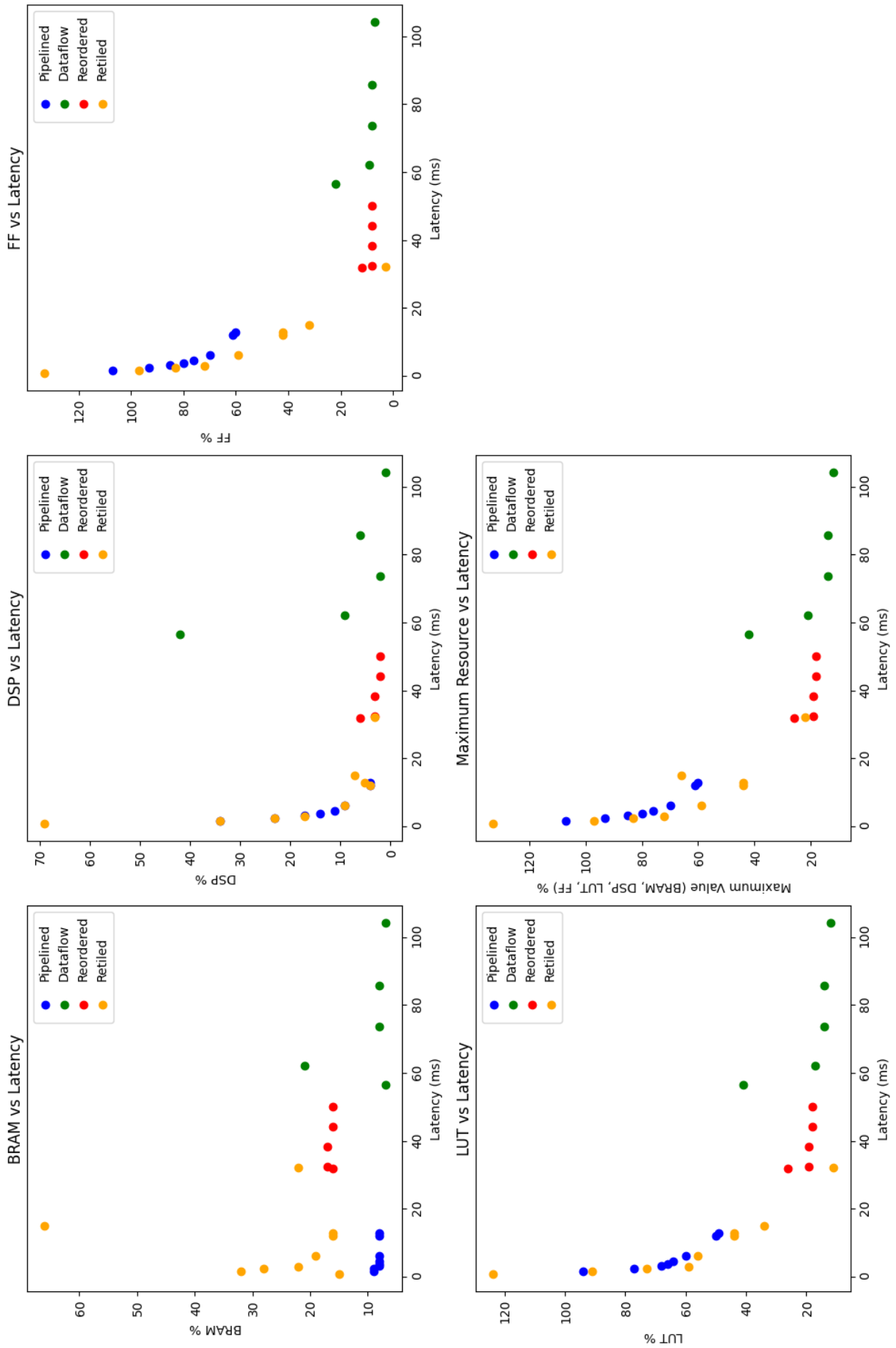


Figure 5.3: Utilisation vs latency of 120 band 13 code autoencoders.

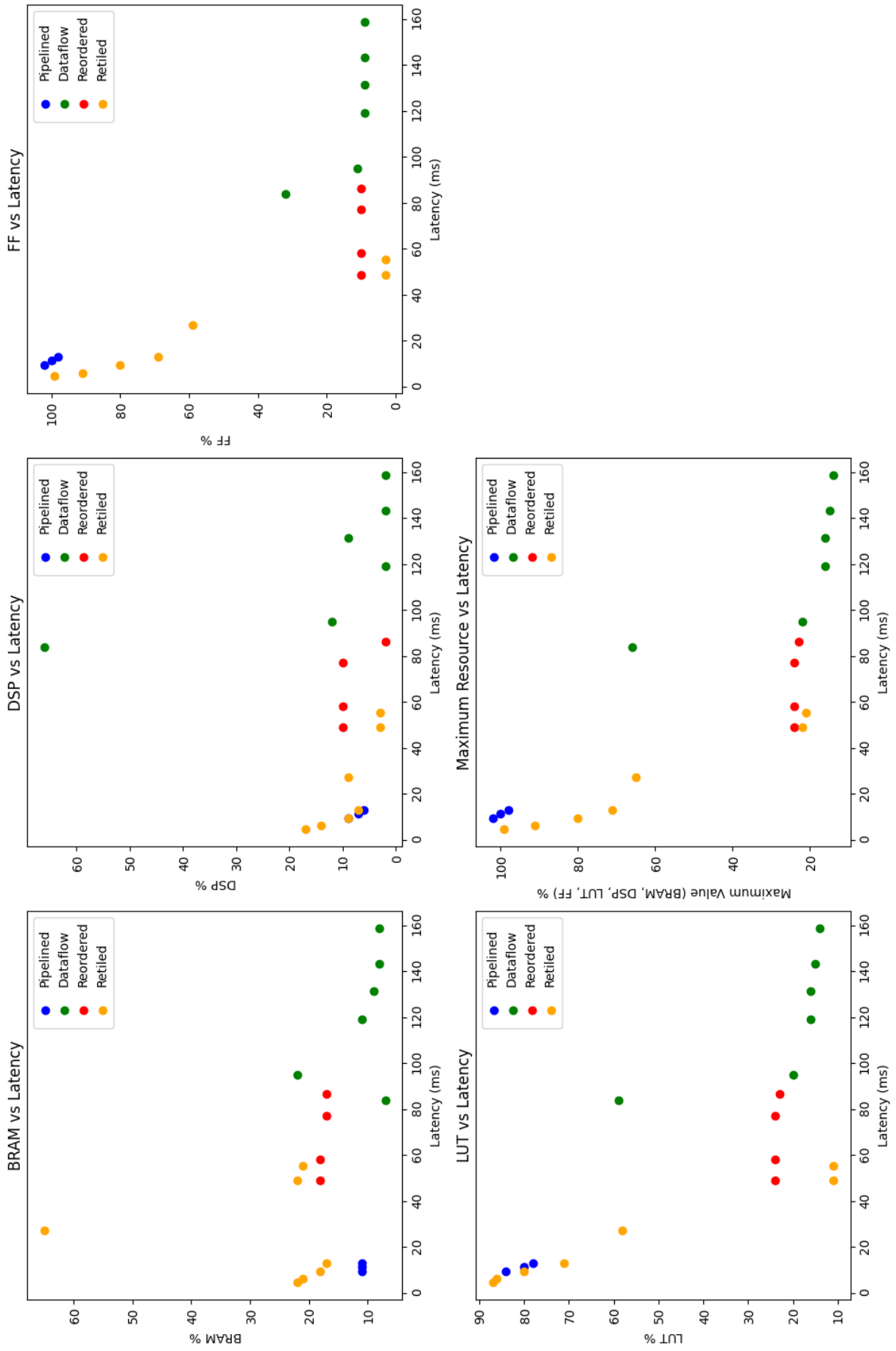


Figure 5.4: Utilisation vs latency of 188 band 13 code autoencoders.

The results provide the performance estimates for each resource of each architecture for the synthesised directives.

- For BRAMs the graphs show that generally the pipelined and dataflow architectures require less BRAM resources for all band code configurations while the retiled in a coarse-grain configuration requires the most. This is harder to interpret as it does not have a clear distinction between pipelined and coarse-grain utilisations.
- For DSPs the retiled and pipelined architectures are almost equal in terms of resource use vs latency, whilst the dataflow and reordered use under 10% utilisation for a range of latencies in each band code size. In the DSP graphs it can be noted that as the band size and increases the utilisation reduces, this is potentially due to the latency increasing.
- FF utilisation is highest for fine-grain pipelined architectures and lower for coarse-grain architectures. The retiled architecture has a lower FF utilisation than the pipelined architectures for the same latency, with reductions up to 30%, indicating they are dominant to these architectures for this resource. In comparison between the dataflow and reordered, the reordered uses more FFs, likely due to it having more computation units as seen in Table 5.1.
- LUT utilisation follows a similar to the FFs, where there is a decrease in utilisation for the retiled at the same latency, with the pipelined architectures consuming more resources.
- The maximum resource utilisation graphs show that the fine-grained pipelined architectures are have higher resource utilisations than the coarse-grain pipelines and this increases significantly with band and code sizes. In contrast the coarse-grain architectures see a slight increase in resources as the band and code sizes increase.

5.1.2 Exploration of Hyperspectral Anomaly Detection with Novel-AWDBN

Following the exploration of AEs described above, the next step is assessing how these implementations fit together with the generated DSWs in Tables 4.20 and 4.21. This is presented to assess the feasibility and achievable throughputs of potential HSI-AD accelerators. The differences in accuracy for full-band ABU images has been presented in Table 4.4. To summarise, for windows that have less neighbours and therefore less operations per pixel, there is generally a decrease in accuracy with less resource utilisation seen for full band ABU images. Therefore there is an element of how window parameters influence achievable throughputs and utilisations combined with AUC score accuracy.

Graphs of AEs coupled with DSWs for different window parameters

The following graphs intend to visualise achievable throughput and resource utilisations of the already synthesised designs when combined modularly. There is some potential for this utilisation to be lower, as the DSW are not throughput optimised for a specific AE and intended to be modular IPs that can work with any AE up to 16 code bands. This exploration stage is performed over each of the four prior established AE band and code combinations. These graphs were generated by entering a desired throughput in cycles per pixel for one of the band code combinations. A sublist of AEs that can meet that throughput requirement is generated from the exploration table in Appendix A. Depending on the throughput requirement, DSW utilisations from Tables 4.20 and 4.21 are added to each AE of the list. The AE-DSW with the lowest maximum resource utilisation that meets the interval requirement is represented as the best-case design that achieves the corresponding throughput.

- HSI-AD Novel-AWDBN is presented for 19 bands, 6 code bands in Fig. 5.5. For cycles per pixel, 120, 224, 480, 600. The corresponding throughput is indicated in the graph. This was generated from Table A.1 combined with Tables 4.20 and 4.21.
- HSI-AD Novel-AWDBN is introduced for 32 bands, 8 code bands in Fig. 5.6. For cycles per pixel, 120, 224, 480, 600. The corresponding throughput is marked in the graph. This was generated from Appendix Table A.2 combined with Tables 4.20 and 4.21.

- HSI-AD Novel-AWDBN is shown for 120 bands, 13 code bands in Fig. 5.7. For cycles per pixel, 120, 240, 480, 685. The corresponding throughput is specified in the graph. This was generated from Appendix Table A.3 combined with Tables 4.20 and 4.21.
- HSI-AD Novel-AWDBN can be seen for 188 bands, 13 code bands in Fig. 5.8. For cycles per pixel, 188, 256, 600, 1024. The corresponding throughput is mentioned in the graph. This was generated from Appendix Table A.4 combined with Tables 4.20 and 4.21.

These designs were not synthesised for these throughputs directly, so they are not the local maxima of optimisation for that given throughput or initation interval, but serve as an indication of the local design space. With further refinements, resources utilised for each throughput category can be reduced.

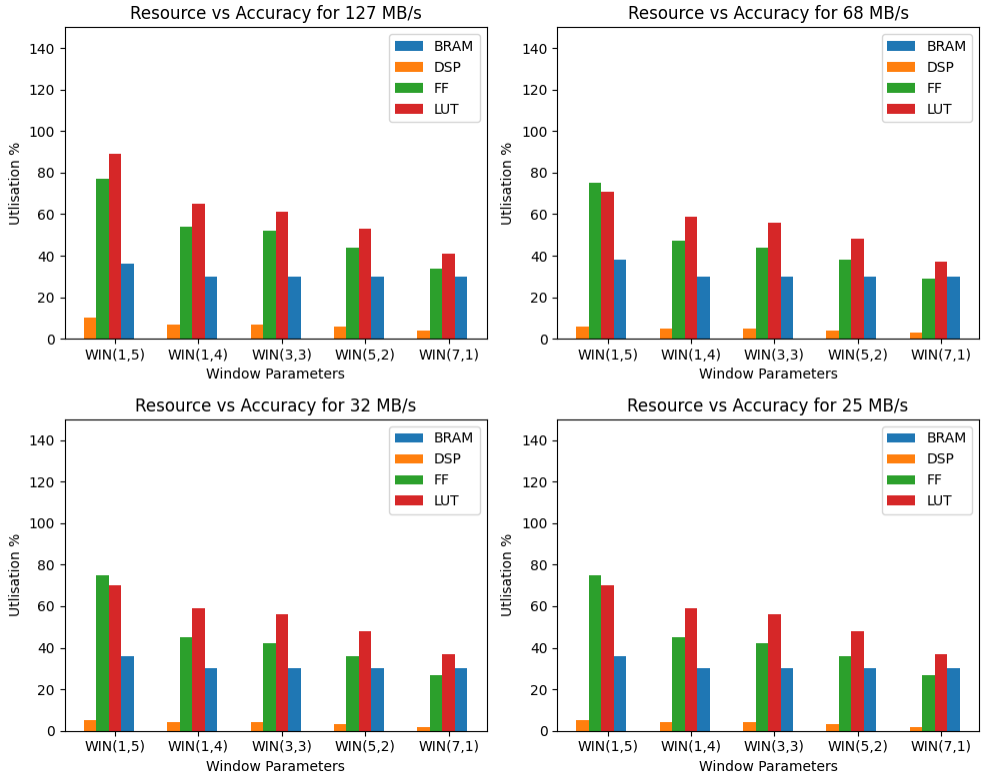


Figure 5.5: Band 19, code size 6, comparison of resources vs throughput for Novel-AWDBN HSI-AD.

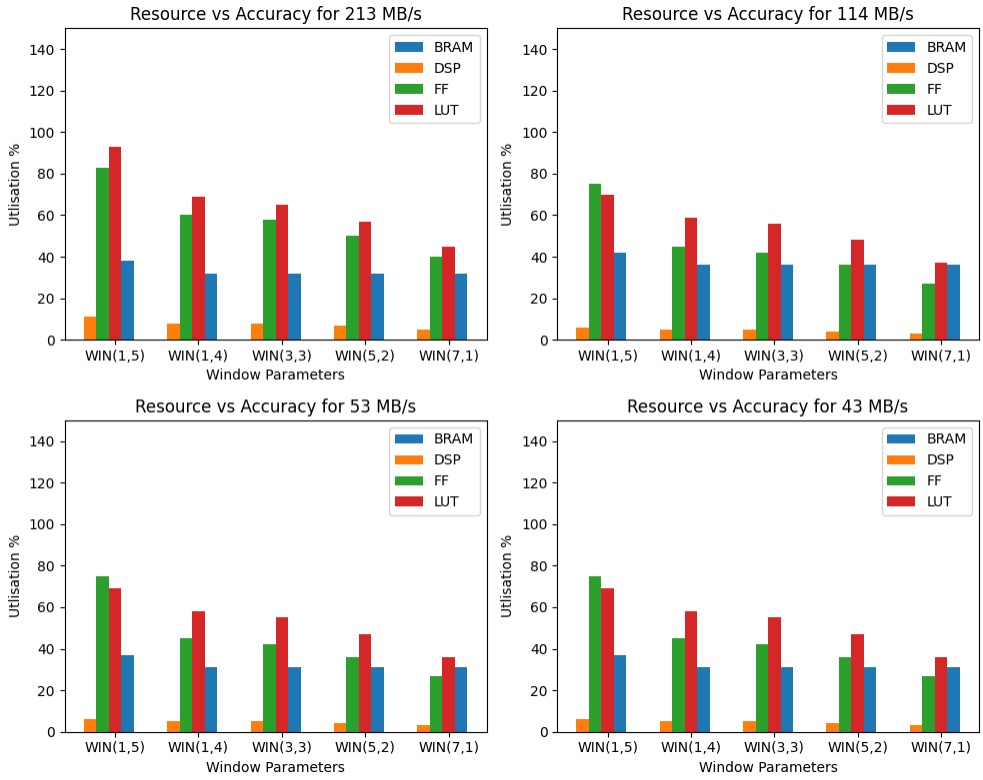


Figure 5.6: Band 32, code size 8, comparison of resources vs throughput for Novel-AWDBN HSI-AD.

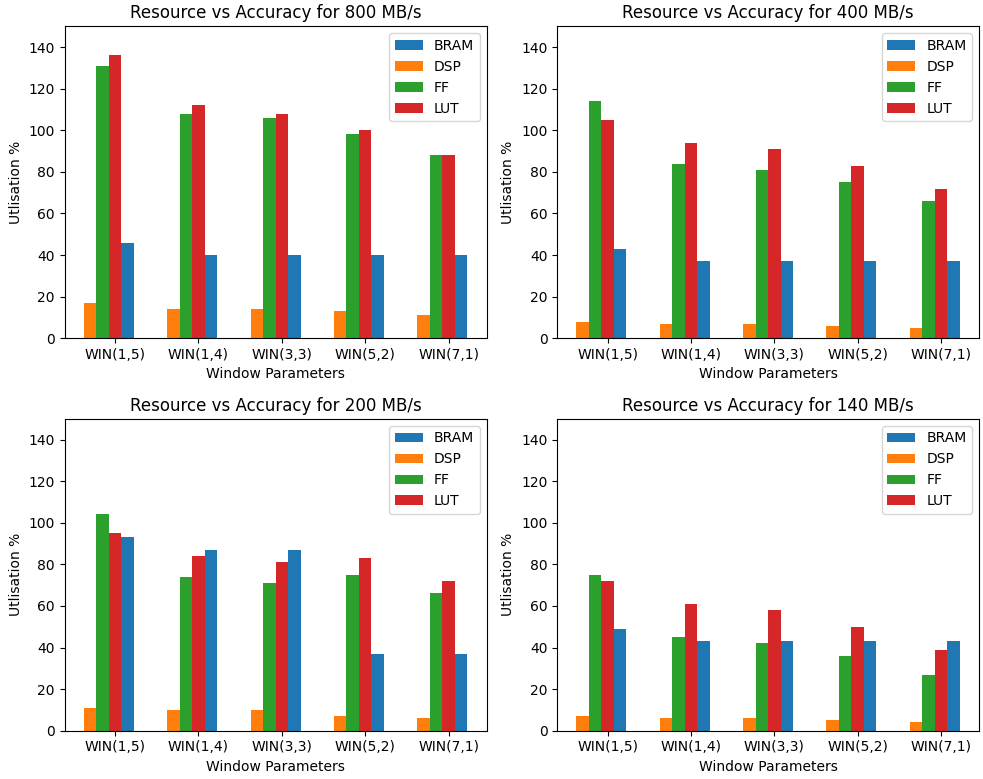


Figure 5.7: Band 120, code size 13, comparison of resources vs throughput for Novel-AWDBN HSI-AD.

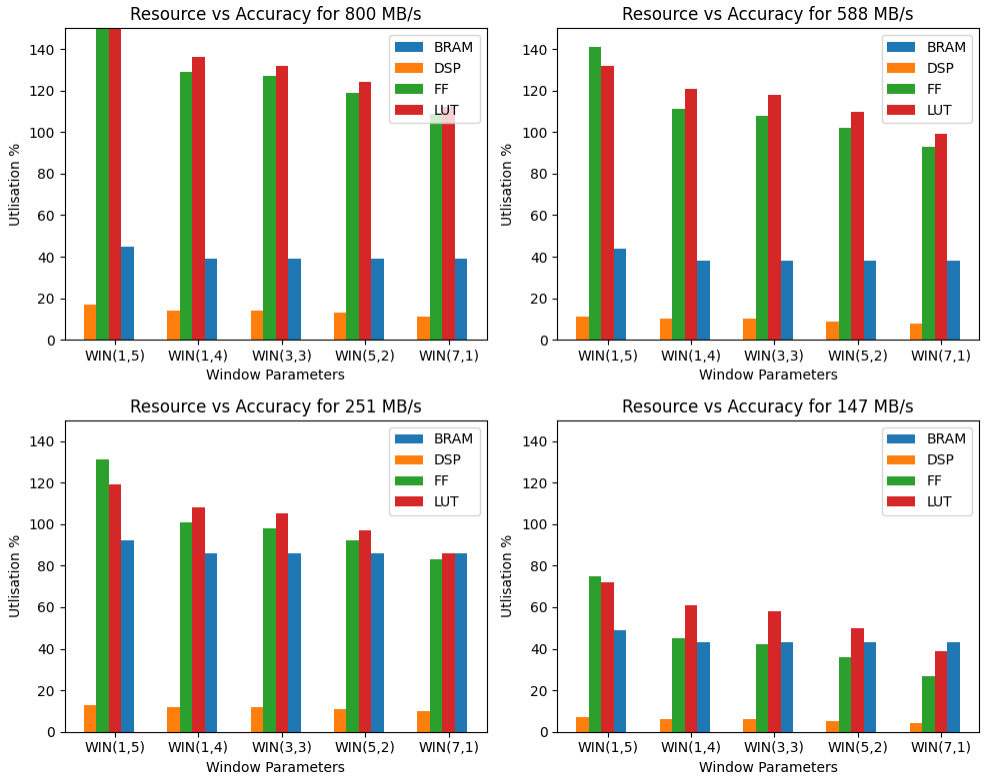


Figure 5.8: Band 188, code size 13, comparison of resources vs throughput for Novel-AWDBN HSI-AD.

In the graphs of the full AWDBN throughputs and window parameters, the following results were found. In most designs, the LUT and FFs were the highest utilised resource while DSPs were the lowest. BRAMs remained relatively constant across all combinations. As band and code size increased the utilisation for each throughput increased but also the throughputs increased as the images are now larger.

5.2 Integration and Testing

In this section results from the integration and physical implementation of two AWDBN accelerators with flexible code and band size. This is to test and evaluate whether previous results are consistent for more generalist accelerators.

5.2.1 Integration of Evaluation Design

To test designs physically on the FPGA the DSW and AE have been integrated through an AXI-Stream interface. Code and reconstruction errors are streamed synchronously from the AE to the DSW. The AE reads pixel and weight data and the DSW now writes data to memory through an AXI-MM interface, which acts as a dedicated DMA. Prior to this integration all of the designs have been modified to allow for flexible band and code sizes, and so a design will be created to accommodate only PCA images and one will be created for up to 120 bands and a code size of 13, which can be used to accelerate larger images or PCA images equally. The block diagram can be seen in Fig. 5.9. This design is in the PL fabric of the ZCU-104 and is connected to the PS by HP ports, one for reading weights and biases and an additional port which allows for parallel receiving and sending of pixel data. These designs are connected by an AXI-Stream internally in Vitis-HLS. There is flexibility for this as they could either both read and write to memory individually, this would be less efficient than streaming due to cycles used in reading and writing. This AXI-Stream implies a dataflow coarse-grain pipelined approach, while the AE is performing the encoding, decoding and RE calculations, the DSW is performing the adaptive weights. Both

units are operating in parallel in this configuration. Both designs share their AXI-Lite registers, which allow the height, width, band size and mid layer code size to be programmable, providing more adaptability in the design as long as it is synthesised to accommodate the maximum of each parameter.

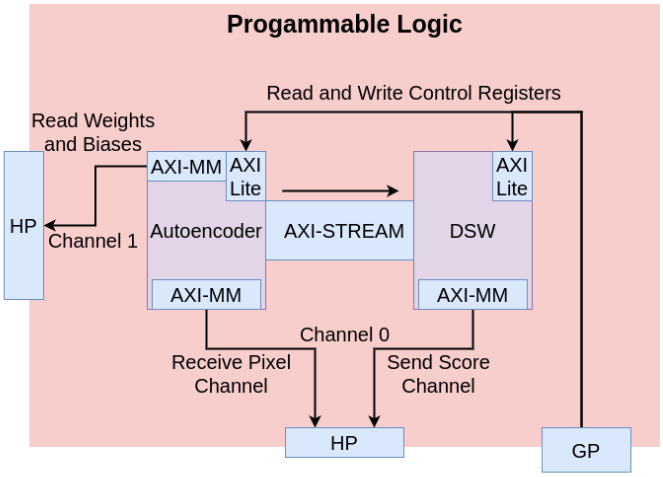


Figure 5.9: Block diagram of integrated implementations.

Changes to image height has no impact on the design utilisation while width only impacts the BRAM utilisation in the image-line buffers of the DSW. Changing code size has implications in both the operations and memory in the DSW and the AE while changing the number of bands can change the size and operations required of the AE only.

5.2.2 Design Parameters

Two designs have been created for testing, shown in Table 5.2.

- A Pipelined, supporting up to 32 bands and 8 mid code layer sizes. Width, Height 150x150.
- B Dataflow, supporting up to 120 bands and code size up to 13. Width, Height 150x150.

Sample Name	Architecture	Max Height, Width	Max Bands	Max Codes	Inner Window	Outer Difference	Frequency	Estimated latency
A PCA	Pipelined	150x150	32	8	7	1	200Mhz	12.8ms
B Full	Dataflow	150x150	120	13	7	1	200Mhz	62ms

Table 5.2: Table of parameters used in the synthesis of the evaluation designs.

These designs have been created for testing that the implementations are able to be synthesised, placed and routed on the FPGA and successfully function when executed physically. They have not been selected to test what is the most suitable or optimised for deployment. They also serve to draw some parallels from the design space exploration in terms of addressing the differences when implemented to accommodate flexibility. Due to the flexible input parameters causing variations in reading and writing internal data streams, the reordered and retiled architectures require more complex FIFO sizing which should be extrapolated through cosimulations to optimise performance and avoid deadlocks which is time prohibitive.

5.2.3 Evaluation

Resource utilisation

Table 5.3 shows the resource utilisations from FPGA implementation in Vivado. Design 1; A PCA which is the pipelined architecture has resources are as expected when compared to the design

space exploration of 32 bands. The 120 band dataflow B design has fared worse in the transition to a flexible architecture.

Sample Name	CLB LUTs	CLB Registers	CLB	LUT Logic	LUT Mem	Block RAM	DSPs
A PCA	35%	20%	60%	30%	10%	1%	6%
B Full	53%	30%	87%	44%	22%	6%	3%

Table 5.3: Table of post-implementation resource utilisation.

C Runtime

The C-AWDBN inference code was cross-compiled in Vitis for execution on the Ultrascale+ ZCU-104 MPSoC. This execution was performed in Linux, executed on a single core of the APU which was operating at 1.2 Ghz. No optimisation flags were specified for this compilation.

Image Height and Width	Window Parameters	Bands	Code size	Runtime (ms)
100 x 100	Inner size; 7, outer difference; 1 (Best case runtime)	19	6	161
		32	8	234
		120	13	848
		188	13	1258
	Inner Size; 1, outer difference; 5 (Worst case runtime)	19	6	284
		32	8	377
		120	13	1035
		188	13	1446

Table 5.4: Runtimes of inference of C AWDBN

Accelerator Execution time

The modules were tested in a bare-metal configuration for three image categories; 19 band, 32 band and 120 bands. Each module has as its input the given weights, biases and image data. The evaluation is performed using Vitis-HLS generated bare-metal drivers and the AXI-Lite status register of the system is polled to test completion of execution. Image data was stored in arrays and in each case written to a buffer before initiating the internal DMA transfer in the accelerator. The runtime results can be seen in Table 5.5 The results show that the A design is able to reduce the execution time up to 94%, with slightly lower execution times as the number of bands and codes are reduced. Design B has a constant execution time for each tested band, code size which is around 10 times higher than Design A but still achieves up to an 84% reduction in execution time for large band sizes.

Configuration			**PCA A (Bands 32, Code 8)**		**Full B (Bands 120, Code 13)**	
Image Bands	Image Code Size	SW Runtime (ms)	HW Runtime (ms)	Speedup %	HW Runtime (ms)	Speedup %
19	6	161	13.8	92%	134	17%
32	8	234	15	94%	134	43%
120	13	848	-	-	134	85%

Table 5.5: Software and accelerator runtime comparison.

5.3 Evaluation

Full AWDBN in C and Matlab

As part of the methodology changes were made to the Novel-AWDBN proposed in work by Gunderson [3]. This included the implementation of AWDBN and Novel-AWDBN into C. Prior work had not yet applied the Novel-AWDBN to the entire image dataset for the trained C AE. This C implemented training had some problems with AWDBN for a code size of 13, to the point it had a lower AUC than just the AE RE scores. The anomaly score was found to improve drastically with a code size of 20, and so the average score is presented in Table 5.6. Weights can be generated in either training implementation and used in the Matlab and C AWDBN inference for equivalent results.

Implementation	Training method	Code size	AUC Average
Matlab DBN	CD & GD	13	0.953797
C DBN	GD	20	0.880633
Matlab or C Novel-AWDBN	From Matlab DBN	13	0.983975
Matlab or C Novel-AWDBN	From C DBN	20	0.964938

Table 5.6: Matlab-AWDBN and full C-AWDBN

Exploration

For further analysis of the exploration results a Pareto frontier has been generated for every band and code size in Fig. 5.10. All designs have a similar shape except for the 188-band. This is influenced by the BRAM utilisation for the design at 30 ms in the 188,13 graph. So the 188 band design has been re-plotted without BRAMs, see Fig. 5.11. Without BRAMs the shape is still different, but slightly closer. Architectures on the Pareto frontier, the red line, are designs that have not had an observed better option for both the objectives throughput and execution time simultaneously. The designs on and closest to the Pareto frontier are the retiled and reordered architectures developed during the AE base architectures section. Although not conclusive this does suggest that they are the most optimal of the designs, this will considered further in the discussion.

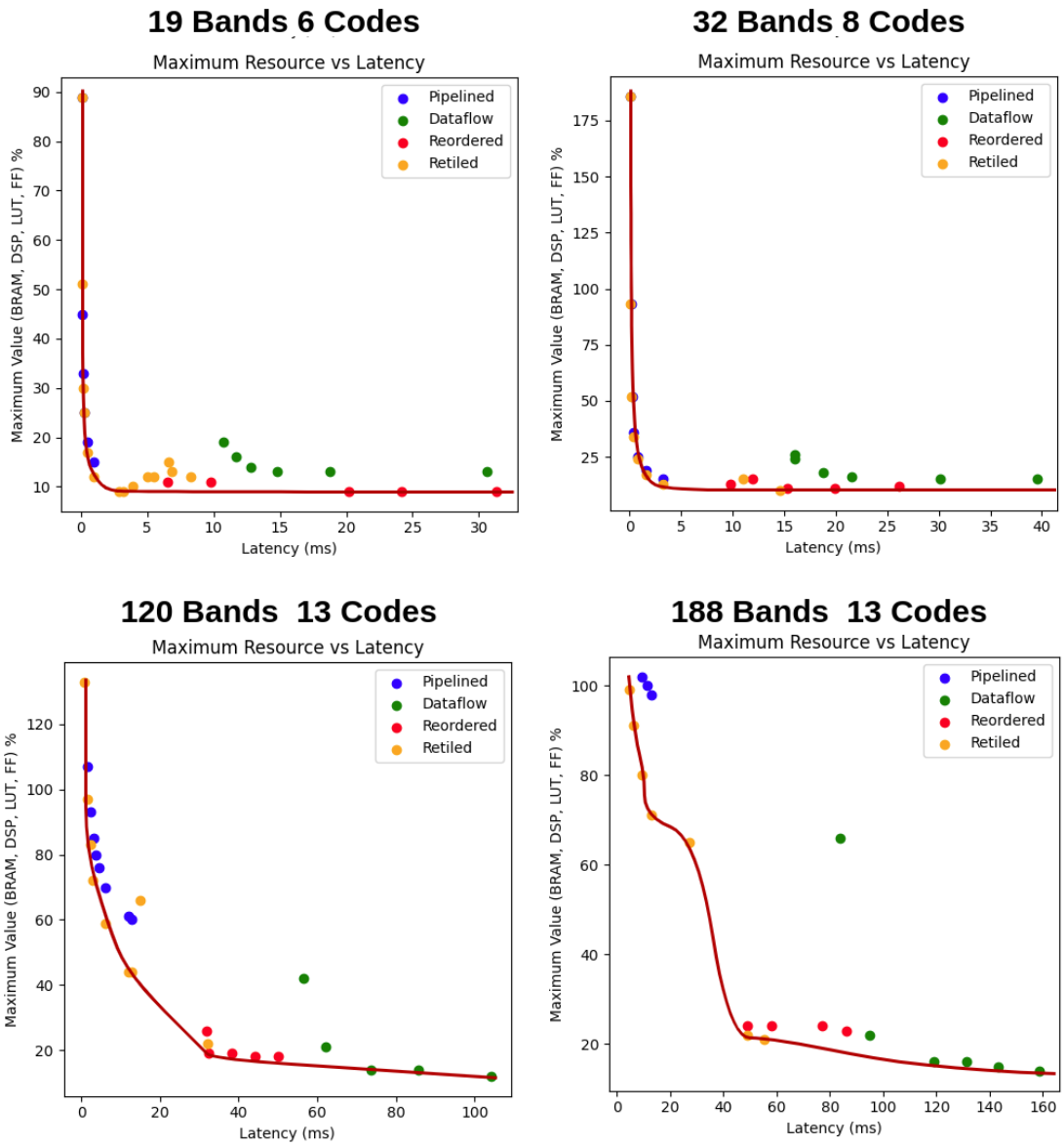


Figure 5.10: Pareto frontiers of explored architectures.

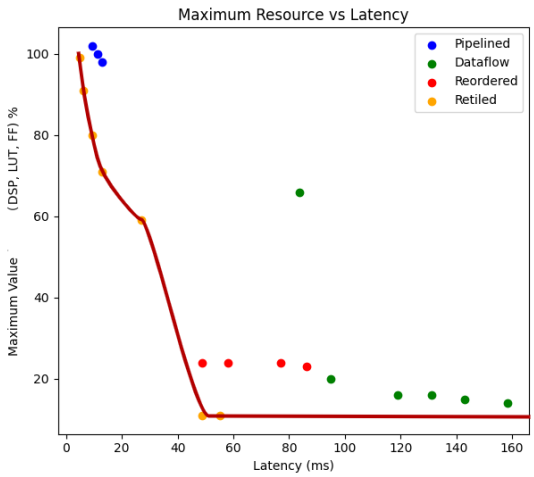


Figure 5.11: Pareto frontier of 188 without BRAM.

Implemented Accelerators

Adaptation to flexibility

Since the exploration was performed for fixed band and code designs, the implemented flexible accelerators can be compared with the design exploration results to see the impact this flexibility has, in Table 5.7. For the A PCA design, flexibility does not seem to have an impact as the resources are lower in the flexible design for a throughput that is not much lower than the fixed exploration. For the 120, 13 Full band design, this design is not close enough in throughput to be truly comparable, but the resource utilisation is similar for a much lower throughput.

Property	Designs		Closest Exploration Design		Exploration vs flexible integrated	
	A PCA	B Full	32,8	120, 13	PCA Accelerator	Full Band Accelerator
Throughput	85 MB/s	36 MB/s	114 MB/s	140 MB/s	74%	26%
LUT	30%	44%	42%	40%	69%	110%
FF	20%	30%	32%	29%	67%	103%
DSP	6%	3%	3%	4%	200%	75%
BRAM / LUT MEM	11%	28%	40%	50%	28%	56%

Table 5.7: Flexible designs and closest fixed exploration design.

Chapter 6

Discussion

In this chapter the results will be considered in more detail and possible explanations for the findings will be explored. How the results can be used to advance knowledge and benefit the HYPSO project will be discussed. Challenges experienced in the design and implementation will be considered and areas for future work will be identified.

6.1 Implications of Results for HYPSO

6.1.1 AWDBN

The project has seen the successful implemnentation of C AWDBN that performs the encoding, decoding and adaptive weights. This was accelerated using an FPGA with the same level of accuracy and a shorter execution time. In theory, with different parameters this hardware accelerator could enable anomoly detection to be performed on the current and future HYPSO satelites.

In terms of the next direction for future work on the AWDBN algorithm itself the focus would now shift to the training and how to utilise these algorithms on HYPSO. For example, the AUC scores for the C trained AWDBN are lower and require a much larger computation complexity at 20 codes compared to 13 codes in Matlab. If the training in C were improved to include CD this could allow for a return to 13 codes, which is at present the least scalable parameter of the accelerator. This leaves the problem that the network is trained for every image and that the training takes up a majority of the computation time. This an overhead of 3.36 seconds for a 188 band image and 1.26 seconds if reduced to 12 bands through PCA. These training times are from prior work by Gunderson [3]. Solving this can be addressed in two different ways. One option is that for AWDBN, either have a larger network trained over a range of images from a ground station, such that the inference can be applied to any image. Or alternatively if training for every image, this would have to be done on HYPSO and as a result the C training will need to be improved and accelerated, or a different less compute intense strategy chosen.

There is also room for improvement in the AWDBN algorithm. Currently only a single window is applied over an entire image. It should be possible to create a heuristic based on surrounding anomalies or penalty distributions to choose a more optimal window size. This means that both small anomalies in images and large anomalies in images will have individual windows tailored to them, therefore increasing the detection accuracy. Looking back at Table 4.2 only a few images shared the same best window size, and there can be varying best window sizes for different regions of any image.

6.1.2 Accelerator Architectures

From the design space exploration, the retiled and reordered typically lie on the pareto frontier while the dataflow and pipeline versions rarely do. This shows that the architecture candidates introduced as potential improvements dominate the pipeline and dataflow counterparts. In some cases the estimated resources of the retiled are lower than the pipelined designs for the same throughput, being as much as 30% lower than the 188-band pipelined for the same throughput. This also applies for the 120-band variants, having an up to 20% reduction in maximum resource utilisation. Additionally different accelerator architectures introduced and explored which have

different amounts of scalability. For any high throughput application the retiled architecture has shown favourable utilisation with the same throughput. On the lower end of utilisation the reordered design has the highest throughput for a similar resource utilisation to the non reordered dataflow at the cost of larger FIFO sizes in BRAM or LUT-RAM. These architectures still require an appropriate FIFO sizing for any given maximum band and code size. This will increase their utilisation but it is still expected they will perform better than the aforementioned counterparts. The general scalability of processing parameters is introduced in Table 6.1. As indicated by the table any foray into a more constrained FPGA such as HYPSO-1 or increasing the network size will require the implementation of the retiled and reordered designs with a FIFO size to suit all band and code parameters.

Paremeter	**Scalability**	**Reasons**	**Future Work**
Image Height	Very high	Only affects execution time.	
Image width	High	Impacts on-chip buffer sizes	Image tiling
Bands	Low in Pipelined and Dataflow	Increase of size and operations in RBM1, RBM2 and Reconstruction Error	Integrate Retiled and Reordered by finding ideal FIFO sizes.
	Medium in Tiled and Reordered	Tiled Bands in RBM1 and Reconstruction Error	
Code Size	Medium∼Low, Depends on window parameters	Increases operations in RBM1, RBM2, memory and size in DSW.	Dataflow DSW

Table 6.1: Scalability of processing parameters in accelerators.

6.1.3 Implemented Accelerator

The accelerator implemented has been shown to be functionally correct through physical testing for custom sizes of bands and codes. For the PCA version of the accelerator there was not a significant variation from the fixed band exploration, however for the 120 band the deviation of throughput at 26% was more significant, see Table 5.7. This throughput level was below estimates for implementation B and is too slow to justify its resource consumption. With further optimisations this variation could be eliminated. Preferably the implementation of architectures on the Pareto frontier of the exploration would see greater improvements. It is also likely there will be a reduction in utilisation from the fixed 120 band with DSW. The DSW module can also be synthesised for a lower amount of code bands at 13 rather than 16. The main preventative factor for the creation of a more impressive and high throughput design has been time limitations. The current implementations are not optimal in terms of architecture choice or fine-tuned in HLS, leaving an opportunity for a large throughput increase with just a small extra effort. Additionally the design should be tested in Linux with appropriate drivers, as this is an important step for creating a design that can be deployed on HYPSO.

6.2 Accelerators for Present and Future HYPSO SmallSats

HYPSO has images of dimension 1000x1000 and 120 bands stored in single precision floating point 32-bit format. This amounts to about 480 MB per image. There are a range of different proposed accelerators at each given throughput, whether or not they would synthesise and implement in Vivado remains to be tested. From the results of this project and the designs that have been tested the following are expected to be possible accelerators for deployment to present and future HYPSOs. Table 6.2 displays potential accelerators that target future HYPSOs that would have a larger FPGA with more resources. Table 6.3 considers HYPSO-1 with the smaller Zynq-7030.

The pre-requisite to these implementations is a working flexible band and code size retiled AE by finding the correct FIFO-Sizing in Vitis-HLS. As an example, the Fig. 5.7 window sizes 7, 1 at 400MB/s should be possible for implementation and handle 0.83 full-band HSIs per second.

Throughput	Window size	120 Band HSIs Per second	Architecture	Max Resource Utilisation	Predicted Synthesisability	Comment
120, 13 at 800 MB/s	7, 1	1.7	Retiled	88%	Congestion and Overutilisation	LUT & FF Pipeline II Low
120, 13 at 400 MB/s	1, 4	0.83	Retiled	94%	Congestion and Overutilisation	LUT & FF
120, 13 at 400 MB/s	3,3	0.83	Retiled	92%	Congestion and Overutilisation	LUT & FF
120, 13 at 400 MB/s	5,2	0.83	Retiled	82%	Overutilisation	LUT & FF
120, 13 at 400 MB/s	7,1	0.83	Retiled	72%	Feasible	Synthesise DSW for 13 bands
120, 13 at 140MB/s	1, 5	0.29	Reordered	76%	Feasible	Synthesise DSW for 13 bands

Table 6.2: Expected synthesisability of AWDBN accelerators for ZCU-104.

However, the FPGA on HYPSO-1 is equivalent in resources to the PicoZed, a ZYNQ-7000 SoC, this has about 30% to 40% of the resources of the ZCU-104.

6.2.1 Accelerator for HYPSO-1

Table 6.3 provides predictions for the feasible accelerators that can be uploaded to HYPSO-1. A pre-requisite of these is the implementation of a reordered architecture. Additionally, if the DSW were synthesised in a dataflow configuration this could reduce the resources and increase the code size scalability further reducing resource use. Of the observed designs from the exploration, comments are made on the likelihood of successful physical implementation on the FPGA in Table 6.3.

Throughput	Window Size	HSIs Per Second	Architecture	Max resource utilisation	Synthesisability	Comment
120, 13 at 140 MB/s	7,1	0.29	Reordered	70∼90%	Uncertain	Synthesise DSW for 13 codes
120, 13 at 140 MB/s	7,1	0.29	Dataflow Retiled	70∼90%	Uncertain	Synthesise DSW for 13 codes

Table 6.3: Expected synthesisability of AWDBN accelerators for HYPSO-1.

6.3 Exploration and Synthesis of Flexible Architectures

A full exploration for the flexible variations of each design was intended, but this became prohibitive due to the time taken to synthesise these designs. The first flexible design prototype had a synthesis time of over 24 hours. This was found to be caused by break statements in the code paired with if statements. The design only consisted of fixed boundary loops as recommended by [36], but had variable controlled break statements. This took a long time to synthesise in HLS, but provided desirable resource estimates once finished. However once implemented on the FPGA this had a large amount of routing issues. This was improved during the integration and implementation chapter by;

- All variable break statements were removed and replaced with conditional assignments of out of bound data, which would be why B-Full implemented design has the same execution time for every band size, it performs the same operations but conditionally manipulates results of calculating codes and bands out of bound to 0.
- All internal streams and communication channels were reduced to 32-bit.
- Pipelining initiation intervals were increased to distribute routing over more clock cycles.

6.3.1 Floating to Fixed Point

Another important step to reduce the utilisation and improve the scalability of accelerators for NNs is a fixed point conversion. This was tested briefly for the proposed architectures, but it was drastically increasing overall resource utilisation. This requires more time and investigation to find out why this is the case and overcome it.

6.4 Future Work

The main points of future work summarised from the discussion are as follows:

- Enhancing the AWDBN algorithm by applying a heuristic to choose an optimal window size based on surrounding anomalies or penalty distributions, thereby improving detection accuracy.
- Exploring strategies to utilise AWDBN on HYPSO by training a larger network over a range of images or a new approach to training which can be performed onboard.
- Investigating appropriate FIFO-sizings for retiled and reordered architectures to allow flexible operation without deadlocks, subsequently implementing and evaluating this.
- Conducting a full exploration and synthesis of flexible variations of accelerator designs.
- Conversion from floating point to fixed point
- Implementation of dataflow-based architecture for DSW.
- Exploring the possibility of using multiple PEs instead of one large pipelined accelerator.
- Testing on Linux with Vitis-HLS generated user-level drivers.
- Tuning and synthesising designs targetting HYPSO-1 specifically.

6.5 Conclusion

In conclusion this thesis refined the adaptive weights of the high level anomaly detection algorithm AWDBN into a low-level C code and the full algorithm further into a hardware implementation. New architectures were successfully explored for the pre-existing implementation of the DBN used in AWDBN. The AW and DBN were successfully integrated to perform a full accelerated inference with up to 85% to 94% reduction in computation time with designs that are not yet fine-tuned or on the Pareto frontier of the undertaken exploration. This leaves a lot of opportunity for improvements at a lower effort margin. Upon analysis of prior work some slight adjustments to the algorithm were made which improved its coverage to provide scoring over an entire HSI rather than missing edges or corners. This change improved the overall image score when considering this uncovered area and, if considering the covered area only, would be equivalent in score. This adjustment is a net positive for all factors except processing time. To make utilising AWDBN more feasible in embedded systems an FPGA implementation in HLS was created for the DSW, which successfully passed physical testing. This has room for further exploration. Many architectures were proposed for the AE and RE portion of the problem and this showed a favourable exploration with new architectures featuring on the Pareto frontier of computation time against resources for this problem. These have yet to be tested and require more co-simulation to find ideal FIFO sizing. These coupled with the DSW had a range of different utilisations at different performance points with the opportunity for fine-tuning to reduce this utilisation further. This has culminated in a summary of possible designs that could be used to accelerate the inference of AWDBN on the ZCU-104 and HYPSO-1. It is estimated that this should be achievable at a throughput of at least 140 MB/s and 70% to 90% resource utilisation on its ZYNQ-7030 FPGA after the FIFO sizing is found. This thesis has provided results that have direct implications for the HYPSO project as well as clearly identifying the areas that future work should focus on.

Bibliography

[1] Mariusz E. Grotte, Roger Birkeland, Evelyn Honore-Livermore et al. 'Ocean Color Hyperspectral Remote Sensing With High Resolution and Low Latency—The HYPSO-1 CubeSat Mission'. In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2022), pp. 1–19. DOI: 10.1109/tgrs.2021.3080175. URL: https://doi.org/10.1109/tgrs.2021.3080175.

[2] Sivert Bakken, Marie B. Henriksen, Roger Birkeland et al. 'HYPSO-1 CubeSat: First Images and In-Orbit Characterization'. In: *Remote Sensing* 15.3 (2023). ISSN: 2072-4292. DOI: 10.3390/rs15030755. URL: https://www.mdpi.com/2072-4292/15/3/755.

[3] Aksel Gunderson. 'Hardware-Software partitioned implementation of an autoencoder- based hyperspectral anomaly detector'. 2021.

[4] Ma Ning, Peng Yu, Wang Shaojun et al. 'A weight SAE based hyperspectral image anomaly targets detection'. In: *2017 13th IEEE International Conference on Electronic Measurement &amp Instruments (ICEMI)*. IEEE, Oct. 2017. DOI: 10.1109/icemi.2017.8265874. URL: https://doi.org/10.1109/icemi.2017.8265874.

[5] Sefa Küçük and Seniha Esen Yüksel. 'Comparison of RX-based anomaly detectors on synthetic and real hyperspectral data'. In: *2015 7th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*. 2015, pp. 1–4. DOI: 10.1109/WHISPERS.2015.8075504.

[6] Peg Shippert. 'Why Use Hyperspectral Imagery?' In: *Photogrammetric Engineering and Remote Sensing* 70 (Apr. 2004).

[7] Đorđije Bošković, Milica Orlandić and Tor Arne Johansen. 'A reconfigurable multi-mode implementation of hyperspectral target detection algorithms'. In: *Microprocessors and Microsystems* 78 (2020), p. 103258. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2020.103258. URL: https://www.sciencedirect.com/science/article/pii/S014193312030418X.

[8] Aksel S. Danielsen, Tor Arne Johansen and Joseph L. Garrett. 'Self-Organizing Maps for Clustering Hyperspectral Images On-Board a CubeSat'. In: *Remote Sensing* 13.20 (2021). ISSN: 2072-4292. DOI: 10.3390/rs13204174. URL: https://www.mdpi.com/2072-4292/13/20/4174.

[9] Andrew Myers. *CubeSat: The little satellite that could*. June 2022. URL: https://engineering.stanford.edu/magazine/cubesat-little-satellite-could.

[10] Qin Ding, Qiang Ding and William Perrizo. 'PARM-An Efficient Algorithm to Mine Association Rules From Spatial Data'. In: *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society* 38 (Jan. 2009), pp. 1513–24. DOI: 10.1109/TSMCB.2008.927730.

[11] I T Jolliffe. *Principal component analysis*. en. 1986th ed. Springer series in statistics. New York, NY: Springer, Mar. 2013.

[12] Xudong Kang. *ABU Dataset*. URL: http://xudongkang.weebly.com/data-sets.html.

[13] I.S. Reed and X. Yu. 'Adaptive multiple-band CFAR detection of an optical pattern with unknown spectral distribution'. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 38.10 (1990), pp. 1760–1770. DOI: 10.1109/29.60107.

[14] Prasanta Chandra Mahalanobis. *On the generalized distance in statistics*. 1936.

[15] Qiong Ran, Zedong Liu, Xiaotong Sun et al. 'Anomaly Detection for Hyperspectral Images Based on Improved Low-Rank and Sparse Representation and Joint Gaussian Mixture Distribution'. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 14 (2021), pp. 6339–6352. DOI: 10.1109/JSTARS.2021.3087588.

[16] Weimin Liu and Chein-I Chang. 'A nested spatial window-based approach to target detection for hyperspectral imagery'. In: *IGARSS 2004. 2004 IEEE International Geoscience and Remote Sensing Symposium*. Vol. 1. 2004, p. 268. DOI: 10.1109/IGARSS.2004.1369012.

[17] Peter Flach, Jose Hernandez-Orallo and Cèsar Ferri. 'A Coherent Interpretation of AUC as a Measure of Aggregated Classification Performance.' In: Jan. 2011, pp. 657–664.

[18] Chein-I Chang. *Hyperspectral imaging*. New York, NY: Springer, July 2003.

[19] Heesung Kwon and N.M. Nasrabadi. 'Kernel RX-algorithm: a nonlinear anomaly detector for hyperspectral imagery'. In: *IEEE Transactions on Geoscience and Remote Sensing* 43.2 (2005), pp. 388–397. DOI: 10.1109/TGRS.2004.841487.

[20] Ran Tao, Xudong Zhao, Wei Li et al. 'Hyperspectral Anomaly Detection by Fractional Fourier Entropy'. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.12 (2019), pp. 4920–4929. DOI: 10.1109/JSTARS.2019.2940278.

[21] Wei Li and Qian Du. 'Collaborative Representation for Hyperspectral Anomaly Detection'. In: *IEEE Transactions on Geoscience and Remote Sensing* 53.3 (2015), pp. 1463–1474. DOI: 10.1109/TGRS.2014.2343955.

[22] Bing Tu, Nanying Li, Zhuolang Liao et al. 'Hyperspectral Anomaly Detection via Spatial Density Background Purification'. In: *Remote Sensing* 11.22 (2019). ISSN: 2072-4292. DOI: 10.3390/rs11222618. URL: https://www.mdpi.com/2072-4292/11/22/2618.

[23] Xudong Kang, Xiangping Zhang, Shutao Li et al. 'Hyperspectral Anomaly Detection With Attribute and Edge-Preserving Filters'. In: *IEEE Transactions on Geoscience and Remote Sensing* 55.10 (2017), pp. 5600–5611. DOI: 10.1109/TGRS.2017.2710145.

[24] Ferdi Andika, Mia Rizkinia and Masahiro Okuda. 'A Hyperspectral Anomaly Detection Algorithm Based on Morphological Profile and Attribute Filter with Band Selection and Automatic Determination of Maximum Area'. In: *Remote Sensing* 12.20 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12203387. URL: https://www.mdpi.com/2072-4292/12/20/3387.

[25] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[26] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio et al. 'Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead'. In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/access.2020.3039858. URL: https://doi.org/10.1109/access.2020.3039858.

[27] Sebastian Ruder. 'An overview of gradient descent optimization algorithms'. In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[28] Ning Ma, Yu Peng, Shaojun Wang et al. 'An Unsupervised Deep Hyperspectral Anomaly Detector'. In: *Sensors* 18.3 (2018). ISSN: 1424-8220. DOI: 10.3390/s18030693. URL: https://www.mdpi.com/1424-8220/18/3/693.

[29] Ning Ma, Shaojun Wang, Yu Peng et al. 'A DBN based anomaly targets detector for HSI'. In: *AOPC 2017: 3D Measurement Technology for Intelligent Manufacturing*. Ed. by Wolfgang Osten, Anand K. Asundi and Huijie Zhao. SPIE, Oct. 2017. DOI: 10.1117/12.2285766. URL: https://doi.org/10.1117/12.2285766.

[30] G. De Michell and R.K. Gupta. 'Hardware/software co-design'. In: *Proceedings of the IEEE* 85.3 (Mar. 1997), pp. 349–365. DOI: 10.1109/5.558708. URL: https://doi.org/10.1109/5.558708.

[31] João M.P. Cardoso, José Gabriel F. Coutinho and Pedro C. Diniz. 'Chapter 8 - Additional topics'. In: *Embedded Computing for High Performance*. Ed. by João M.P. Cardoso, José Gabriel F. Coutinho and Pedro C. Diniz. Boston: Morgan Kaufmann, 2017, pp. 255–280. ISBN: 978-0-12-804189-5. DOI: https://doi.org/10.1016/B978-0-12-804189-5.00008-9. URL: https://www.sciencedirect.com/science/article/pii/B9780128041895000089.

[32] Stephen M. Trimberger. 'Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology'. In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104.

[33] Scott Hauck and Andre Dehon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Jan. 2008. Chap. 31.

[34] Louise H Crockett, David Northcote and Craig Ramsay. *Exploring zynq MPSoC*. Strathclyde Academic Media, Apr. 2019.

[35] Michael Fingeroff. *High-Level synthesis blue book*. Alamo, TX: Xlibris, May 2010.

[36] Xilinx. *Vitis High-Level Synthesis User Guide*. Dec. 2021. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug1399-vitis-hls.pdf.

[37] Xilinx. *Vivado Design Suite User Guide: Getting Started (UG910)*. Oct. 2021. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug910-vivado-getting-started.pdf.

[38] Xilinx. *Vitis Unified Software Platform*. 2019. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[39] Xilinx. *Zynq® UltraScale+™ MPSoC*. 2015. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[40] Xilinx. *ZCU104 Evaluation Board User Guide (UG1267)*. Oct. 2018. URL: https://www.xilinx.com/support/documents/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[41] Xilinx. *AMBA AXI4 Interface Protocol*. URL: https://www.xilinx.com/products/intellectual-property/axi.html.

[42] ARM. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*. Feb. 2013. URL: https://developer.arm.com/documentation/ihi0022/e/.

[43] Xilinx. *Vivado Design Suite: AXI Reference Guide (UG1037)*. July 2017. URL: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide.

[44] Maria Angelopoulou and Christos Bouganis. 'Vision-Based Egomotion Estimation on FPGA for Unmanned Aerial Vehicle Navigation'. In: *IEEE Transactions on Circuits and Systems for Video Technology* 24 (June 2014), pp. 1–1. DOI: 10.1109/TCSVT.2013.2291356.

[45] Onur Ulusel, Christopher Picardo, Christopher B. Harris et al. 'Hardware acceleration of feature detection and description algorithms on low-power embedded platforms'. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577310.

[46] Manish Kumar Jaiswal and Hayden K.-H. So. 'DSP48E efficient floating point multiplier architectures on FPGA'. In: *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. 2017, pp. 1–6. DOI: 10.1109/ICVD.2017.7913322.

[47] Ambrose Finnerty and Hervé Ratigner. 'Reduce power and cost by converting from floating point to fixed point'. In: *WP491 (v1. 0)* (2017).

[48] Rishabh Goyal, Joaquin Vanschoren, Victor van Acht et al. *Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms*. 2021. arXiv: 2102.02147 [cs.CV].

Appendices

Appendix A

Exploration Tables

- X indicates unrolled and inlined.
- II=[%d] indicates pipelined at initiation interval d.
- II=[%d]X indicates unrolled and pipelined at initiation interval d.
- - indicates not applicable
- I indicates integrated into RBM2
- MEM = MIX/AUTO means automatic memory object implementation by Vitis-HLS
- Partition = AUTO means automatic memory partitioning by Vitis-HLS
- Partition = %d means that Weights for RBM1 and Weights for RBM2 were block partitioned at factor d, and the rest of the implementation uses automatic partitioning.

Bands	Codes	Arch	Interval	RBM1	RBM1-I	RBM1-0	RBM2	RBM2-I	RBM2-O	RE	Port_Width	Latency	BRAM	DSP	FF	LUT	URAM	MEM	PARTITION
19	6	Dataflow	612	II=64	X	-	II=28	X	-	II=36	32	6121488	3	1	7	13	0	MIX	AUTO
19	6	Dataflow	375	II=32	X	-	II=14	X	-	II=18	32	3,751,208	3	1	7	13	0	MIX	AUTO
19	6	Dataflow	295	II=16	X	-	II=8	X	-	II=12	32	2951001	3	2	7	13	0	MIX	AUTO
19	6	Dataflow	255	II=8	X	-	II=4	X	-	II=6	32	2560909	4	2	7	14	0	MIX	AUTO
19	6	Dataflow	232	II=4	X	-	II=2	X	-	II=3	32	2330869	4	3	8	16	0	MIX	AUTO
19	6	Dataflow	215	II=1	X	-	II=14	X	-	II=1	32	2150964	7	9	10	19	0	MIX	AUTO
19	6	Pipelined	19	X	X	-	X	X	-	I	32	190897	3	5	10	15	0	MIX	AUTO
19	6	Pipelined	10	X	X	-	X	X	-	I	64	100541	3	9	14	19	0	MIX	AUTO
19	6	Pipelined	5	X	X	-	X	X	-	I	128	50542	3	18	17	25	0	MIX	AUTO
19	6	Pipelined	3	X	X	-	X	X	-	I	256	31705	1	30	21	33	0	MIX	AUTO
19	6	Pipelined	2	X	X	-	X	X	-	I	512	21699	5	45	27	43	0	MIX	AUTO
19	6	Pipelined	1	X	X	-	X	X	-	I	1024	11699	1	89	40	67	0	MIX	AUTO
19	6	Reordered	627	II=200	X	II=200	II=200	X	II=32	II=200	32	6270172	9	1	3	9	0	MIX	AUTO
19	6	Reordered	483	II=200	X	II=200	II=200	X	II=24	II=200	32	4830182	9	1	3	9	0	MIX	AUTO
19	6	Reordered	404	II=100	X	II=100	II=100	X	II=16	II=20	32	4040173	9	1	3	9	0	MIX	AUTO
19	6	Reordered	195	II=3	X	II=14	II=8	X	II=8	II=8	32	1950172	11	2	3	10	0	MIX	AUTO
19	6	Reordered	131	II=1	X	II=1	II=1	X	II=1	II=1	32	1310173	10	2	4	11	0	MIX	AUTO
19	6	Retiled	165	II=7	X	II=2	II=2	X	-	II=160X	32	1651537	9	2	5	12	0	MIX	AUTO
19	6	Retiled	137	II=3	X	II=3	II=1	X	-	II=137X	64	1371568	9	3	6	13	0	MIX	AUTO
19	6	Retiled	132	II=3	X	II=3	II=1	X	-	II=130X	128	1320100	9	3	6	15	0	MIX	AUTO
19	6	Retiled	110	II=1	X	II=3	II=100X	X	-	II=100X	32	1100257	9	2	6	12	0	MIX	AUTO
19	6	Retiled	100	II=100X	X	II=3	II=100X	X	-	II=100X	32	1001895	9	2	7	12	0	MIX	AUTO
19	6	Retiled	78	II=77X	X	II=3	II=77X	X	-	II=77X	32	780968	9	2	5	10	0	MIX	AUTO
19	6	Retiled	64	II=64X	X	II=1	II=64X	X	-	II=64X	32	640931	9	2	5	9	0	MIX	AUTO
19	6	Retiled	58	II=57X	X	II=1	II=57X	X	-	II=57X	32	580957	9	2	5	9	0	MIX	AUTO
19	6	Retiled	19	II=19X	X	II=19X	II=19X	X	-	II=19X	32	190994	11	5	10	12	0	MIX	AUTO
19	6	Retiled	10	II=10X	X	II=10X	II=10X	X	-	II=10X	64	100981	9	9	13	17	0	MIX	AUTO
19	6	Retiled	5	II=5X	X	II=5X	II=5X	X	-	II=5X	128	51191	9	18	18	25	0	MIX	AUTO
19	6	Retiled	4	II=3X	X	II=3X	II=3X	X	-	II=3X	256	41199	9	26	21	30	0	MIX	AUTO
19	6	Retiled	2	II=2X	X	II=2X	II=2X	X	-	II=2X	512	20935	23	45	34	51	0	MIX	AUTO
19	6	Retiled	1	II=1X	X	II=1X	II=1X	X	-	II=1X	1024	10927	27	89	50	77	0	MIX	AUTO

Table A.1: Table of exploration of 19 band 6 code AE.

Bands	Codes	Arch	Interval	RBM1	RBM1-I	RBM1-0	RBM2	RBM2-I	RBM2-O	RE	Port_Width	Latency	BRAM	DSP	FF	LUT	URAM	MEM	PARTITION
32	8	Dataflow	II=790	II=64	X	-	II=20	X	-	II=36	32	7902471	4	1	13	15	0	MIX	AUTO
32	8	Dataflow	II=602	II=32	X	-	II=10	X	-	II=18	32	6031943	4	1	13	15	0	MIX	AUTO
32	8	Dataflow	II=431	II=16	X	-	II=5	X	-	II=9	32	4311564	4	2	13	16	0	MIX	AUTO
32	8	Dataflow	II=376	II=8	X	-	II=5	X	-	II=6	32	3761725	8	2	14	18	0	MIX	AUTO
32	8	Dataflow	II=321	II=1	X	-	II=4	X	-	II=4	32	3211923	3	11	18	24	0	MIX	AUTO
32	8	Dataflow	II=321	II=1	X	-	II=1	X	-	II=1	32	3212080	3	13	20	26	0	MIX	AUTO
32	8	Pipelined	II=64	X	X	-	X	X	-	I	32	643400	4	3	14	15	0	MIX	AUTO
32	8	Pipelined	II=32	X	X	-	X	X	-	I	32	323416	4	6	17	19	0	MIX	AUTO
32	8	Pipelined	II=16	X	X	-	X	X	-	I	64	163424	4	11	23	25	0	MIX	AUTO
32	8	Pipelined	II=8	X	X	-	X	X	-	I	128	80850	5	23	30	36	0	MIX	AUTO
32	8	Pipelined	II=4	X	X	-	X	X	-	I	256	43416	6	46	38	52	0	MIX	AUTO
32	8	Pipelined	II=2	X	X	-	X	X	-	I	512	23415	6	93	54	83	0	MIX	AUTO
32	8	Pipelined	II=1	X	X	-	X	X	-	I	1024	13414	1	186	80	128	0	MIX	AUTO
32	8	Reordered	II=523	II=16	X	II=32	II=13	X	II=13	II=13	32	5230310	12	2	4	12	0	MIX	AUTO
32	8	Reordered	II=399	II=12	X	II=16	II=10	X	II=10	II=10	32	3990311	11	2	4	11	0	MIX	AUTO
32	8	Reordered	II=306	II=9	X	II=12	II=8	X	II=8	II=8	32	3060341	11	2	4	11	0	MIX	AUTO
32	8	Reordered	II=239	II=7	X	II=12	II=6	X	II=6	II=6	32	2390346	15	2	5	13	0	MIX	AUTO
32	8	Reordered	II=196	II=1	X	II=1	II=1	X	II=1	II=1	32	1960325	11	3	5	13	0	MIX	AUTO
32	8	Retiled	II=291	II=8	X	II=16	II=4	X	-	II=8	32	2910320	10	2	3	8	0	MIX	AUTO
32	8	Retiled	II=222	II=6	X	II=12	II=2	X	-	II=4	32	2200316	15	2	3	9	0	MIX	AUTO
32	8	Retiled	II=64	II=64X	X	II=64X	II=64X	X	-	II=64X	32	641351	11	3	11	13	0	MIX	AUTO
32	8	Retiled	II=32	II=32X	X	II=32X	II=32X	X	-	II=64X	32	321081	12	6	15	17	0	MIX	AUTO
32	8	Retiled	II=16	II=16X	X	II=16X	II=16X	X	-	II=16X	64	161066	15	11	22	24	0	MIX	AUTO
32	8	Retiled	II=8	II=8X	X	II=8X	II=8X	X	-	II=8X	128	81054	10	23	30	34	0	MIX	AUTO
32	8	Retiled	II=4	II=4X	X	II=4X	II=4X	X	-	II=4X	256	41024	11	46	39	52	0	MIX	AUTO
32	8	Retiled	II=2	II=2X	X	II=2X	II=2X	X	-	II=2X	512	21010	23	93	56	84	0	MIX	AUTO
32	8	Retiled	II=1	II=1X	X	II=1X	II=1X	X	-	II=1X	1024	11003	27	186	102	152	0	MIX	AUTO

Table A.2: Table of exploration of 32 band 8 code AE

Bands	Codes	Arch	Interval	RBM1	RBM1-I	RBM1-0	RBM2	RBM2-I	RBM2-O	RE	Port_Width	Latency	BRAM	DSP	FF	LUT	URAM	MEM	PARTITION
120	13	Dataflow	2081	II=75	X	-	II=16	X	-	II=16	32	20819165	7	1	7	12	0	MIX	AUTO
120	13	Dataflow	1715	II=40	X	-	II=13	X	-	II=13	32	17159955	8	6	8	14	0	MIX	AUTO
120	13	Dataflow	1472	II=30	X	-	II=10	X	-	II=12	32	14729595	8	2	8	14	0	MIX	AUTO
120	13	Dataflow	1249	II=13	X	-	II=9	X	-	II=9	32	12409146	21	9	9	17	0	MIX	AUTO
120	13	Dataflow	1128	II=1	X	-	II=8	X	-	II=8	32	11300443	7	42	22	41	0	MIX	AUTO
120	13	Dataflow	1090	II=1	X	-	II=1	X	-	II=1	32	10910842	56	46	23	44	0	MIX	AUTO
120	13	Pipelined	256	X	X	-	X	X	-	I	32	2578837	8	4	60	49	0	MIX	AUTO
120	13	Pipelined	240	X	X	-	X	X	-	I	32	2413054	8	4	61	50	0	MIX	AUTO
120	13	Pipelined	120	X	X	-	X	X	-	I	32	1212854	8	9	70	60	0	MIX	AUTO
120	13	Pipelined	90	X	X	-	X	X	-	I	64	912912	8	11	76	64	0	MIX	AUTO
120	13	Pipelined	75	X	X	-	X	X	-	I	64	768646	8	14	80	66	0	MIX	AUTO
120	13	Pipelined	60	X	X	-	X	X	-	I	64	618572	8	17	85	68	0	MIX	AUTO
120	13	Pipelined	45	X	X	-	X	X	-	I	128	468748	9	23	93	77	0	MIX	AUTO
120	13	Pipelined	30	X	X	-	X	X	-	I	128	314580	9	34	107	94	0	MIX	AUTO
120	13	Reordered	1003	II=8	X	II=60	II=50	X	II=8	II=8	32	10031827	16	2	8	18	0	MIX	AUTO
120	13	Reordered	884	II=7	X	II=45	II=45	X	II=7	II=7	32	8841827	16	2	8	18	0	MIX	AUTO
120	13	Reordered	764	II=6	X	II=40	II=40	X	II=6	II=6	32	7651700	17	3	8	19	0	MIX	AUTO
120	13	Reordered	646	II=5	X	II=30	II=30	X	II=5	II=5	32	6461827	17	3	8	19	0	MIX	AUTO
120	13	Reordered	636	II=5	X	II=1	II=5	X	II=1	II=5	32	6361716	16	6	12	26	0	MIX	AUTO
120	13	Retiled	643	II=1	X	II=45	II=3	X	-	II=1	32	6438358	22	3	3	11	0	MIX	AUTO
120	13	Retiled	299	II=256X	X	II=20	II=1	X	-	II = 256X	32	2993825	66	7	32	34	0	MIX	4
120	13	Retiled	256	II=256X	X	II=256X	II=256X	X	-	II = 256X	32	2581846	16	5	42	44	0	MIX	4
120	13	Retiled	240	II=240X	X	II=240X	II=240X	X	-	II=240X	32	2404805	16	4	42	44	0	MIX	4
120	13	Retiled	120	II=120X	X	II=120X	II=120X	X	-	II=120X	32	1209097	19	9	59	56	0	MIX	8
120	13	Retiled	60	II=60X	X	II=60X	II=60X	X	-	II=60X	64	608927	22	17	72	59	0	MIX	13
120	13	Retiled	45	II=45X	X	II=45X	II=45X	X	-	II=45X	128	454160	28	23	83	73	0	MIX	20
120	13	Retiled	30	II=30X	X	II=30X	II=30X	X	-	II=30X	128	302445	32	34	97	91	0	MIX	27
120	13	Retiled	15	II=15X	X	II=15X	II=15X	X	-	II=15X	256	153821	15	69	133	124	0	MIX	54

Table A.3: Table of exploration of 120 band 13 code AE

Bands	Codes	Arch	Interval	RBM1	RBM1-I	RBM1-0	RBM2	RBM2-I	RBM2-O	RE	Port_Width	Latency	BRAM	DSP	FF	LUT	URAM	MEM	PARTITION
188	13	Dataflow	3170	II=120	X	-	II=16	X	-	II=16	32	31704304	8	2	9	14	0	AUTO	AUTO
188	13	Dataflow	2861	II=100	X	-	II=14	X	-	II=14	32	28613614	8	2	9	15	0	AUTO	AUTO
188	13	Dataflow	2624	II=80	X	-	II=13	X	-	II=13	32	26245684	9	9	9	16	0	AUTO	AUTO
188	13	Dataflow	2361	II=60	X	-	II=11	X	-	II=11	32	23814936	11	2	9	16	0	AUTO	AUTO
188	13	Dataflow	1900	II=20	X	-	II=9	X	-	II=9	32	19014189	22	12	11	20	0	AUTO	AUTO
188	13	Dataflow	1674	II=1	X	-	II=8	X	-	II=8	32	16746260	7	66	32	59	0	AUTO	AUTO
188	13	Pipelined	256	X	X	-	X	X	-	X	32	2564432	11	6	98	78	0	AUTO	AUTO
188	13	Pipelined	224	X	X	-	X	X	-	X	32	2268823	11	7	100	80	0	AUTO	AUTO
188	13	Pipelined	188	X	X	-	X	X	-	X	32	1884324	11	9	102	84	0	AUTO	AUTO
188	13	Reordered	1726	II=9	X	II=2	II=96	X	II=5	II=9	32	17272672	17	2	10	23	0	MIX	AUTO
188	13	Reordered	1539	II=8	X	II=2	II=80	X	II=4	II=8	32	15392672	17	10	10	24	0	MIX	AUTO
188	13	Reordered	1163	II=6	X	II=2	II=80	X	II=3	II=6	32	11632659	18	10	10	24	0	MIX	AUTO
188	13	Reordered	976	II=1	X	II=1	II=64	X	II=4	II=1	32	9762726	18	10	10	24	0	MIX	AUTO
188	13	Retiled	1106	II=3	X	II=9	II=4	X	-	II=3	32	11064591	21	3	3	11	0	AUTO	AUTO
188	13	Retiled	976	II=1	X	II=8	II=3	X	-	II=1	32	9762667	22	3	3	11	0	AUTO	AUTO
188	13	Retiled	540	II=256X	X	II=1	II=1	X	-	II=256X	32	5405195	65	9	59	58	0	AUTO	5
188	13	Retiled	256	II=256X	X	II=256X	II=256X	X	-	II=256X	32	2565932	17	7	69	71	0	AUTO	5
188	13	Retiled	188	II=188X	X	II=188X	II=188X	X	-	II=188X	32	1886111	18	9	80	80	0	AUTO	7
188	13	Retiled	120	II=120X	X	II=120X	II=120X	X	-	II=120X	64	1205874	21	14	91	86	0	AUTO	11
188	13	Retiled	94	II=94X	X	II=94X	II=94X	X	-	II=94X	64	945851	22	17	99	87	0	AUTO	13

Table A.4: Table of exploration of 188 band 13 code AE

NTNU

Norwegian University of
Science and Technology