NTNU
Kunnskap for en bedre verden

DEPARTMENT OF ELECTRONIC SYSTEMS

TFE4940 - ELECTRONIC SYSTEMS DESIGN AND INNOVATION,
MASTER'S THESIS

FPGA Accelerated Convolution Layer Implementation for Semantic Segmentation of Hyperspectral Images

Author:
Magnus Ramsfjell

May 2023

Abstract

Remote sensing using satellites has become an important step in environmental monitoring. The HYPSO Mission aims to detect harmful algal bloom from space using the HYPSO-1 satellite. It is equipped with a hyperspectral imager capable of capturing images with incredible spectral detail. These images are large — too large to mindlessly transmit to earth every time they are taken. Thus, the use of machine learning on the satellites internal instruments is motivated. Convolutional Neural Networks (CNNs) are have been shown to be excellent at performing image segmentation. Using such a network on a hyperspectral imager in orbit can classify the images at minimum energy utilisation. This thesis aims to develop an efficient and fast convolutional layer to be used in neural networks for semantic image segmentation of hyperspectral images on the satellite's on-board FPGA. High-Level Synthesis was used for development. The implementation gave satisfactory performance numbers, with a full convolution layer taking approximately 1.8ms to compute for a 512×512 image with 10 spectral bands.

Sammendrag

Avstandsobservering med satellitter har vært et viktig steg i miljøovervåkning. HYPSO-oppdraget forsøker å detektere skadelig algevekst fra verdensrommet med hjelp av satellitten HYPSO-1. Den er utstyrt med et hyperspektral kamera med egenskaper til å ta bilder med rikt spektralt innhold. Disse bildene er store — for store til å overføre til basestasjoner hver gang de blir tatt. Derfor er det ønskelig å bruke maskinlæring for prosessering av bildene med satellittens interne prosesseringssystemer. Konvolusjonelle nevrale nettverk har vist seg nyttige til segmentering av bilder. Satellitten kan bruke slike nettverk til å klassifisere informasjon i hyperspektrale bilder med lavt strømforbruk. Denne oppgaven utvikler et rask og energieffektiv konvolusjonelt lag for bruk i segmentering av hyperspektrale bilder direkte på satellittens innebygde FPGA. High-Level Synthesis blir brukt til utvikling. Implementasjonen ga tilfredsstillende ytelse, med om lag 1.8ms for å utføre konvolusjon på et $512 \times 512 \times 10$ stort hyperspektralt bilde.

Table of Contents

List of Figures

Acronyms

ANN	Artificial Neural Network. 5, 6
ASIC	Application-Specific Integrated Circuit. 9
BIL	Band Interleaved by Line. 4, 13, 15
BIP	Band Interleaved by Pixel. 4
BSQ	Band SeQuential. 4
CNN	Convolutional Neural Network. 6–9, 12, 21
CONOPS	Consept of Operation. v, 2
DR	Dynamic Reconfiguration. 3
FPGA	Field-Programmable Gate Array. 9, 13, 21
GDA	Generalised Discriminant Analysis. 8
HAB	Harmful Algeal Bloom. 2
HDL	Hardware Definition Language. 15
HLS	High-Level Synthesis. 10, 11, 15
HSI	Hyperspectral Imaging. 3
HYPSO	Hyper-Spectral Small Satellite for Ocean Observation. 2
IC	Integrated Circuit. 9
II	Initiation Interval. 15, 20
LDA	Linear Discriminant Analysis. 8
LWIR	Long-wave Infrared. 3
ML	Machine Learning. 4–6
PCA	Principal Component Analysis. 8
PL	Programmable Logic. 10, 13
PS	Processing System. 9, 10, 13, 15, 21
RTL	Register-Transfer Level. 10, 11, 16

1 Introduction

Satellites in orbit provide an invaluable opportunity to perform observations of the earth's surface. With advances in low-power reconfigurable computational devices, performing in-orbit complex processing has become more and more common. Combine these technological advances and you get the HYPSO Mission. Launched in 2022, HYPSO-1 currently captures valuable hyperspectral images of among other things the North Sea's oceans, which can be used to detect harmful oceanic growth. Several great hurdles have been overcome on the journey to where HYPSO-1 is currently, and the next will be tackled in this thesis.

Semantic image segmentation is the task of assigning a semantic label to each pixel in an image, such as sky, road, tree, etc. Hyperspectral images are images that capture a large number of spectral bands, ranging from visible to infrared wavelengths, and provide rich information about the scene. However, hyperspectral images also pose challenges for semantic segmentation, such as high dimensionality, noise, and spectral variability. Convolutional neural networks (CNNs) are powerful models that can learn hierarchical features from images and achieve state-of-the-art results in semantic segmentation. However, CNNs also require high computational and memory resources, which are limited on-board a satellite. Field-programmable gate arrays (FPGAs) are reconfigurable hardware devices that can offer high performance and low power consumption for CNNs. However, designing and optimising CNNs for FPGAs is not trivial and requires specialised skills and tools. This thesis implements an efficient convolution layer for use in CNNs on an FPGA. The main contributions of this thesis are: (1) a multi-core convolution design tailored for hyperspectral image segmentation and can handle variable number of bands; (2) a method of efficiently importing network weights into a convolutional layer for high reusability of hardware; and (3) discussion on future work for the design. The motivation for this thesis is to enable the detection and classification of certain patterns in the hyperspectral images on-board a satellite with an on-board FPGA and hyperspectral imager to efficiently pick out what data to spend a tight downlink transmission budget on.

2 Background

This chapter gives an insight into various topics related to semantic image segmentation of hyperspectral images. They are presented to explain the design choices taken during implementation. The HYPSO Mission is briefly explained first to motivate further theory. Secondly, the principles of hyperspectral imaging is explained, followed by machine learning theory. The machine learning theory includes an overview of general artificial neural networks before diving deeper into convolutional neural networks. Afterwards follows a brief explanation of Field-Programmable Gate Arrays. The section is concluded by an overview of High-Level Synthesis.

2.1 The HYPSO Mission

The HYPSO (Hyper-Spectral Small Satellite for Ocean Observation) mission aims to observe ocean colour and detect Harmful Algeal Bloom (HAB). This is currently achieved by the HYPSO-1 nanosatellite operated by the SmallSat team at NTNU. The CubeSat includes a hyperspectral imaging payload to perform earth observation and a processing system to perform in-orbit processing. Since its launch with the SpaceX Transporter-3 mission in January 2022 HYPSO-1 has been in a low-Earth orbit of around 500km. HYPSO-2 is a planned nanosattelite with various upgrades from HYPSO-1. It will operate in addition to it's older sibling. The HYPSO-1 CONOPS (Consept of Operation) is visualised in figure 1, and includes five main operations:

- Receive telecommands and other updates from nearby ground station. Orient the hyperspectral imager to start scan of a pre-defined area.
- Execute slew manoeuvre to orient the imager correctly while imaging.
- Processing of images after imaging. Reduce data size for downlink.
- Downlink to nearby ground station.
- Additional supporting assets (e.g. NTNU operated UAVs) may collect additional data if they are in the area.

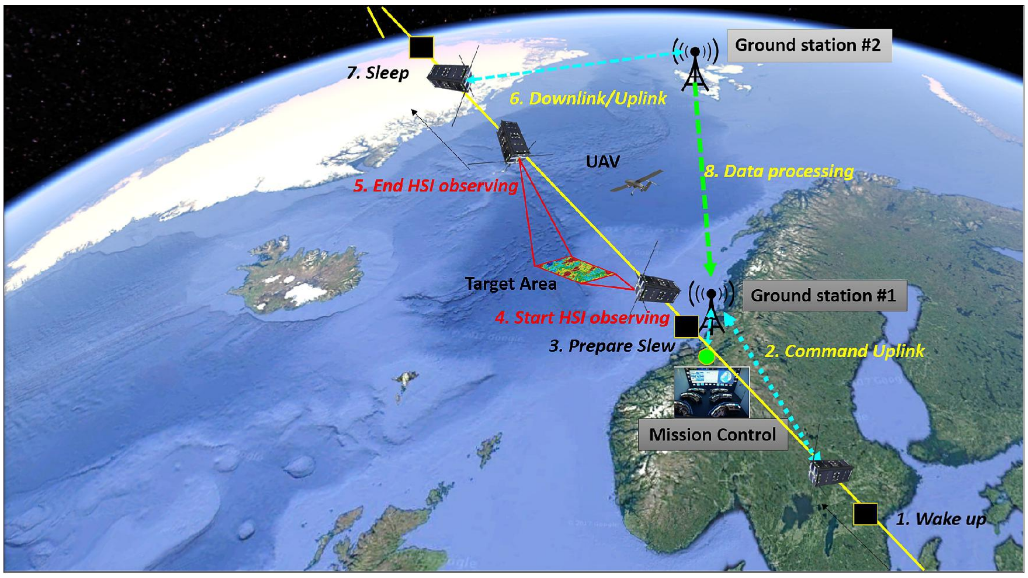


Figure 1: HYPSO CONOPS, where 1) uplinked configuration from ground station is received, 2) hyperspectral imaging is performed, 3) onboard processing occurs, 4) downlink is performed to nearby ground station, and 5) close-by assets can be deployed to gather additional data[1].

While most components and operations of the satellite are finalised, over-the-air updates are still possible for its software components. In particular, processing of images to reduce the data size for downlink is an active area of development. Using machine learning to perform semantic image segmentation (explained in Section 2.3.1) of hyperspectral images can aid in highlighting interesting regions, drastically reducing data downlink size. Downlink is often one of the bottlenecks of small low-earth orbit satellites, and as such it is ideal to reduce the data size. The HYPSO-1 satellite is equipped with a Zynq-7030 FPGA from Xilinx, capable of Dynamic Reconfiguration (DR) This enables the processing system of the satellite to receive functional upgrades, such as improved HAB-detection though semantic image segmentation.

2.2 Hyperspectral Imaging

2.2.1 Remote Sensing

Remote sensing is the acquisition of information about an object without coming into physical contact with it [2]. It typically deals with acquisition, processing, and interpretation of data obtained from sensing a remote object using an image sensor. Hyperspectral Imaging (HSI) is an advanced form of remote sensing utilising many narrow bands of the electromagnetic spectrum.

While mainstream consumer cameras normally capture a scene with three wide spectral bands for red, green, and blue, additional information can be extracted from the scene by using a higher amount of thinner bands. These bands can be contained within the human-visible spectrum, or extend beyond, depending on the electromagnetic characteristics of the scene and the sensing requirements. It may seem intuitive to a human that all visual information about an object is retrievable entirely with their eyesight, but what a human perceives is simply a representation of the human-visible spectrum in their mind. In reality, the interaction between matter and electromagnetic radiation is more complex, showcased in Figure 2 by hyperspectral remote sensing's ability to distinguish between different materials.

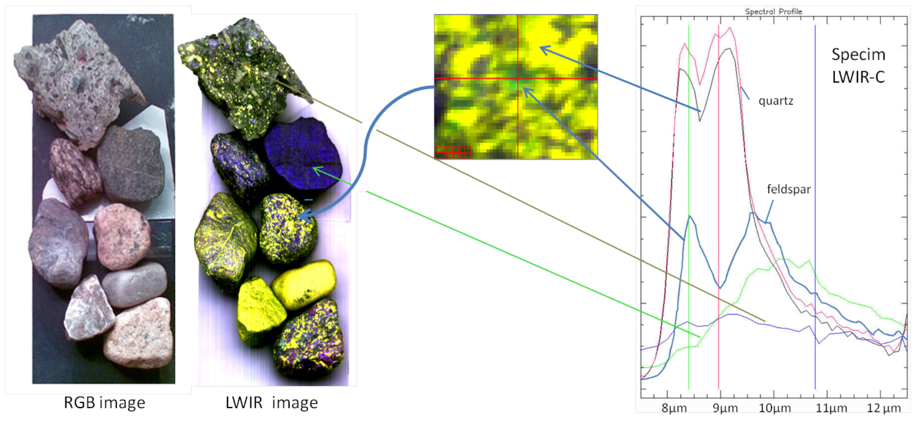


Figure 2: RGB and hyperspectral images of minerals showcasing the latter's ability to ability to distinguish different materials[3].

Figure 2 shows a set of stones which are scanned in the Long-wave Infrared (LWIR) range from 7.7 µm to 12.4 µm. Quartz and feldspar can easily be distinguished by their spectral profiles. This remote sensing capability extends far beyond geology, though, and HSI is often used in conjunction with aerial photography. This can enable an aeroplane or satellite, such as the HYPSO-1, to capture rich information about large areas of land and sea.

2.2.2 Hyperspectral Images

Hyperspectral imaging captures many spectral bands, often hundreds to even thousands. The hyperspectral image is often referred to as an image cube. The height and width of the cube is relatable to a regular image, but the depth is determined by the number of bands. Where regular RGB images consist of three colour bands (red, green, and blue) and overlay its three monochromatic colour bands on top of each other to create a single polychromatic image suitable for human vision, hyperspectral image cubes have too much data to be observed sufficiently in the same manner (altough they can be visualised as RBG images by joining multiple contiguous bands to "recreate" the primary colours of an RBG image). The cubes are usually processed digitally before interesting data emerge.

The most common methods for encoding multiband raster images in the geospacial domain such as hyperspectral images are Band Interleaved by Pixel (BIP), Band Interleaved by Line (BIP), and Band SeQuential (BSQ). The difference between the encoding formats is how the spectral information is stored. BIP will interleave the spectral bands with each spatial pixel, effectively giving all data for one location of the scene at a time. This is similar to how a bitmap raster image would store an RGB-image, giving all colour data for one pixel at a time. BIL differs by first storing one line —one full length in the spatial dimension— of a single band before interleaving the next spectral dimension's data of the same spatial line. After all the spectral band data is stored for the one line, the next is stored, and so on. BSQ encodes multiband images one spectral band at a time. It encodes data in the entire spatial region for one band, line by line, before encoding the next band. The BIL image format is visualised by a 5-by-5-by-5 cube in Figure 3. The numbers on the cells indicate the indices of the corresponding cell after image encoding. The front face represents the data for the first band of the entire 5-by-5 spatial region. In isolation, it could create a monochromatic image of the spatial region. As seen in the figure, the "top" line of the cube is encoded first, followed by the second spectral band of the same spatial line. This continues until all spectral information about the first line is encoded, after which the second line follows.

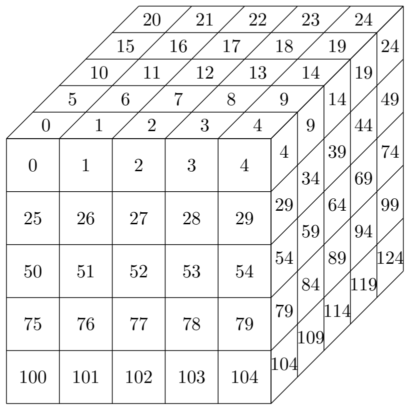


Figure 3: BIL format visualisation, showing how a $5 \times 5 \times 5$ hyperspectral cube is ordered.

The BIL encoding format is favourable for pushbroom-scanning hyperspectral imagers, such as the imager on the HYPSO-1. The pushbroom technique scans one spatial line for all spectral bands before moving on to the next. This is an advantageous technique for a scanner with velocity oriented the same way as the scan direction, as the scanner itself only has to capture spectral information for one spatial dimension at a time. Tilting the scanner in a slew manoeuvre as displayed in 1 results in better spatial resolution, as explained in [4].

2.3 Machine Learning

Machine Learning (ML) is a subset of artificial intelligence that enables computers to learn from data and apply that knowledge in various ways. Machine learning is used to develop programs that can perform a task by the means of learning rather than explicit directives. Image recognition is a common application of machine learning, where neural networks are often used.

2.3.1 Semantic Image Segmentation

Semantic image segmentation is a method that analyses the data of its input, usually an image, and outputs labels corresponding to specific spatial locations of the input. It has the ability to recognise shapes from groups of pixels in an image. It uses machine learning constructs to analyse collections of labelled images, creating a semantic segmentation network used to classify parts of images into categories.

2.3.2 Artificial Neural Networks

An Artificial Neural Network (ANN), or just a neural network, is a type of machine learning algorithm that is inspired by the structure and function of the human brain. It consists of interconnected nodes or neurons that process information and learn from data. The goal of an ANN is to learn a function that maps inputs to outputs by adjusting the weights of the connections between neurons. In the context of image recognition, neural networks can take in a set of input pixels representing an image, and output a probability distribution over a set of possible labels, indicating which object or category the image, or parts of the image, belongs to.

The processing that takes place within a neuron involves summing up the weighted inputs and passing the result through an activation function. The activation function introduces non-linearity into the network, allowing it to model complex relationships between inputs and outputs[5]. The output of a neuron is determined by its activation level, which represents how strongly it is triggered by the input pattern. Neural networks can be trained under supervised learning by being presented with large datasets of labelled data, with which it adjusts its weights to minimise the difference between its predicted outputs and the true outputs. One method of adjusting parameters is known as backpropagation and involves calculating the gradient of the loss function with respect to the weights and updating them accordingly. Unsupervised learning is when the network is trained on unlabelled data. The training process typically involves the following steps: Initialisation, forward propagation, error calculation, backward propagation and adjustment of weights and biases. This process is normally repeated many times to achieve high accuracy when making predictions on new, unseen examples.

A disadvantage of ANNs is that they can be computationally expensive for large datasets. Processing of a 2D-image has every pixel contributing one node in each layer of the network. This means that a relatively small, greyscale image of dimensions 64x64 will require $64 \times 64 \times n_{+textlayers}$ weights for n_{layers} amount of layers, adding up to over 8000 weights for a tiny network of two layers. This is a compounding problem for ANNs when subjected to multidimensional inputs (such as hyperspectral images), as each dimension increase will induce a proportionally large increase in the amount of weights.

In a neural network, a node is activated when the output of that node is above a specified threshold value. The output of a node is computed by applying an activation function to the weighted sum of its inputs. The inputs to a node are the outputs of other nodes in the previous layer of the network, and the weights associated with these inputs determine the strength of their influence on the node's output. If the output of a node is above its threshold value, it sends data to the next layer of the network. Otherwise, no data is passed along to the next layer. The activation function used by a node can be linear or non-linear and determines how the node responds to its inputs. The process can be modelled by Equation 1.

$$y = f(\sum_i w_i x_i - T) \tag{1}$$

For the above equation, y represents node activation, $f()$ is the activation function, w_i is the weight of input x_i, and T is the threshold value[5]. One example of an activation function is the step function, which activates y only if

$$\sum_i w_i x_i > T$$

and otherwise stays inactive. Non-linear transfer function are often more useful for ML applications than linear ones[6][5], and two particularly useful activation functions are the Sigmoid (Equation 2) and ReLU (Equation 3) functions.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of neural networks that utilise the mathematical operation of convolution in an attempt to achieve greater network performance at lower computational costs compared to other neural networks for certain ML workloads[7][8][9]. They retain the advantages of ANNs through self-optimising and self-learning neurons. The neurons are activated in a similar way to ANN neurons by performing operations on its inputs followed by some activation function. Thus, it retains the strengths of ANNs and the classification techniques developed for ANNs apply to CNNs as well.

Convolutional neural networks are defined by having one or more convolutional layers. These layers perform feature detection by convolving matrices of small kernels with the input. They enable CNNs to operate on datasets with large dimensionality without an inhibitably large set of trainable parameters. The relatively small kernel is applied to the entire spacial area of the input, benefitting from reusability where traditional ANNs would require individual neurons for every datapoint of the input. Convolution layers output feature maps which are used by subsequent layers. CNNs typically also have pooling and fully-connected layers[9]. Pooling reduces the spatial dimensions of the feature maps, resulting in fewer trainable parameters, which helps combat overfitting[10]. Fully-connected layers perform classification of the dimensionally reduced feature maps, and produce the final output. Classification is analogous to the fully connected layers of an ANN as explained in Section 2.3.2.

2.4.1 2D Convolutional Layers

Convolution in the field of machine learning refers to the mathematical expression of cross-correlation. The operations involved in the mathematical definition of convolution and cross-correlation are the same, but the kernel is flipped for the former. For the sake of clarity, the operation of cross-correlation is used form this point on, and is simply referred to as convolution, in accordance with the field. The definition for discrete 2D convolution is shown in 4.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, k+n)K(m,n) \tag{4}$$

The resulting feature map $S(i,j)$ is obtained by convolving the 2D matrices representing the kernel $K(i,j)$ and image $I(i,j)$. An example kernel of size $K_s = 3$ is shown in Figure 4. The kernel cells are labelled with their corresponding indices $k_{i,j}$. The convolution operation can be visualised by imagining a window of the same dimensions as the kernel overlaid on the image centred at one spatial location at a time. The window and kernel is point-wise multiplied and accumulated to produce one value to the feature map with a location matching the centre pixel of the window. The size of the output feature map is determined by $(I_s + 2p - K_s/s) + 1$, where I_s is the image size, K_s is the kernel size, s is the stride, and p is padding. The stride is how far the window moves across the window per iteration. A stride of one will produce the largest output, as the kernel will be convolved with every possible window of the image. The situation when the window encounters an edge of the image can be treated in various ways. The window can slide across the image while remaining completely confined within it, causing a reduction in size of the output equal to half the kernel size, rounded down. Dimensionality can be preserved by extending the image outside of its original borders, known as padding. The padded numbers are typically zeroes, but nearest-neighbour or wrapping values from the opposite side can also be added.

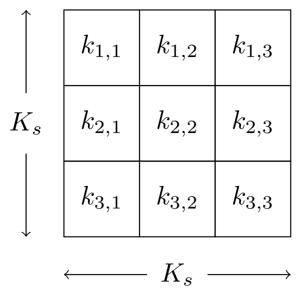
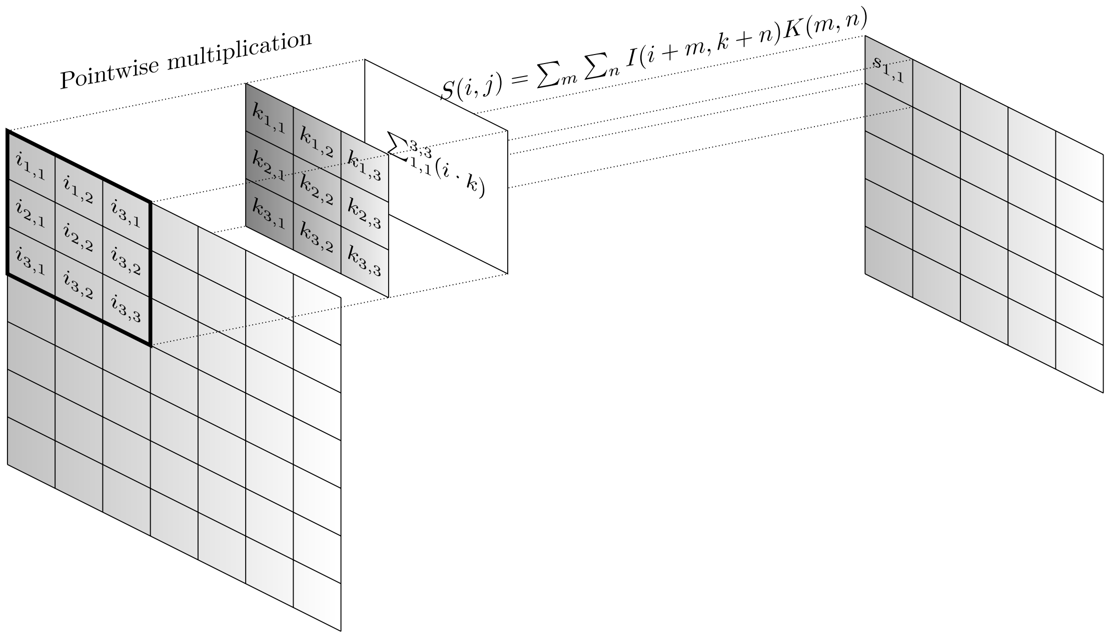

$k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,1}$ $k_{3,2}$ $k_{3,3}$
K_s
K_s

Figure 4: Kernel of size 3×3 visualised

One step of convolution between a 7×7 input image and a 3×3 kernel is shown in Figure 5. There is no padding in the visualisation, and with a stride of one the output feature map is of size 5×5. A sliding window is shown centred over the second row and column of the input image, which is the starting point when there is no padding. The nine selected pixels are point-wise multiplied with their corresponding kernel values, which are then accumulated and stored to the indicated output location. The next step of the convolution operation would be to slide the window across the input image and perform the same arithmetic operation on the newly selected set of values.

The kernel weights are optimised for pattern recognition during training. Multiple kernels are used to detect different patterns in the input data. One convolutional layer of a CNN thus consists of several kernels, effectively adding a dimension of depth to the kernel. Each kernel is convolved with the input image to create an associated feature map.

The output of the first set of kernel convolutions and corresponding activation functions can be used as input to another convolution layer. Adding multiple convolution layers like this can increase the network's ability to recognise complex shapes and patterns[7][9].



Pointwise multiplication
$S(i,j) = \sum_m \sum_n I(i+m, k+n)K(m,n)$
$\sum_{1,1}^{3,3}(i \cdot k)$
$i_{1,1}$ $i_{1,2}$ $i_{3,1}$
$i_{2,1}$ $i_{2,2}$ $i_{3,2}$
$i_{3,1}$ $i_{3,2}$ $i_{3,3}$
$k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,1}$ $k_{3,2}$ $k_{3,3}$
$s_{1,1}$

Figure 5: One iteration of a convolution operation without padding between an image I and kernel K of sizes 7×7 and 3×3, respectively. The output is a 5×5 feature map S.

2.4.2 Depthwise Separable Convolution Layers

A depthwise separable convolution layer in a CNN is a type of convolution operation that is composed of two separate operations: depthwise convolution and pointwise convolution. Depthwise convolution applies a single convolutional kernel per input channel, while pointwise convolution uses a 1×1 convolution to combine the output of the depthwise convolution across channels[11].

Depthwise separable convolution layers are widely used in CNNs because they have two main advantages over standard convolution layers. Firstly, they have fewer parameters to adjust, which reduces overfitting. Secondly, they are computationally cheaper due to fewer computations[11].

2.4.3 Pooling Layers

Pooling layers are a common type of layer that is typically added after convolutional layers in a CNN. The purpose of a pooling layer is to reduce the spatial dimensions of the feature maps while preserving the spectral depth. This downsampling of feature maps makes the resulting downsampled feature maps more robust to changes in the position of the feature in the image, referred to as local translation invariance[12].

Two common pooling methods are average pooling and max pooling. Average pooling computes the average of the current view of the feature map, while max pooling selects the maximum value of the view, thus propagating the most activated presence of a feature within patches of the feature map. Pooling layers provide an approach to downsampling feature maps by summarising the presence of features in patches of the feature map.

There are two groups of pooling layers commonly used in CNNs. Local pooling performs pooling on small regions of the input image to downsample the feature maps[12]. A pooling layer is generally local unless explicitly stated otherwise. The other group is global pooling, where the entire input is pooled to create scalar values of a feature vector.

2.4.4 Transposed 2D Convolution Layers

Network layers like 2D convolutional and pooling layers can only reduce the spatial dimensions of an input or keep them unchanged. It is favourable to produce an output of the same dimensions as the input, as semantic segmentation usually classifies patterns at a pixel-level. This is where transposed convolution layers come in. They reverse the downsampling incurred by previous layers. The operation is visualised for a 2×2 kernel and input in Figure 6. The kernel is sliding with a stride of one over the two-element wide input, for each of the two rows, producing $2 \times 2 \times 2 \times 2$ intermediary results in a 3×3 wide space. The intermediary results are added to produce the output of higher dimensionality than the original input.

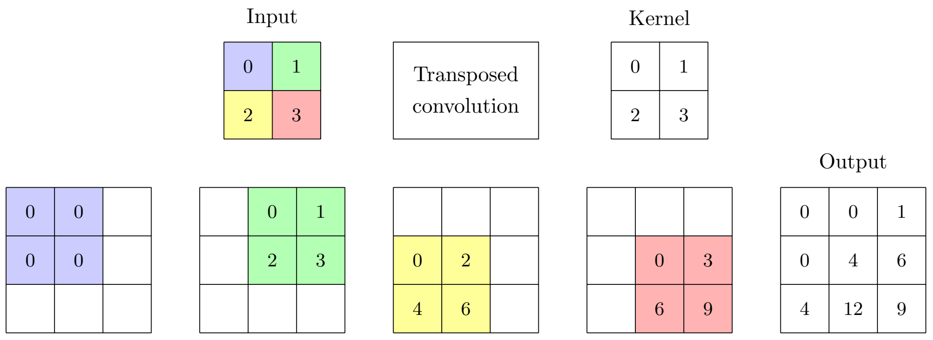


Figure 6: Transposed convolution between a 2×2 kernel and input, producing a 3×3 output. Visualisation inspired by [13].

2.4.5 Concatenation Layers and Skip Connections

Skip connections are used in CNN architectures to connect the output of one layer to the input of another that does not immediately follow it. This allows the network to bypass layers, which can be particularly useful in retaining spatial information when spacial dimensions are reduced though non-padded convolution or pooling layers.

Concatenation layers have the ability to combine multiple input layers into a single output feature map. Used in conjunction with skip connections, concatenation layers are useful to mitigate certain training problems such as vanishing training gradients[14].

2.4.6 Dimensionality Reduction

Hyperspectral images pose a challenge to neural networks due to the computational complexity caused by their high dimensionality. Dimensionality reduction methods can be used to reduce spectral redundancy, which can result in less processing time and enhanced classification accuracy[15][16]. Some of the most commonly used include Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), and Generalised Discriminant Analysis (GDA)[17]. These techniques project the data onto a lower-dimensional space while preserving important information. Dimension reduction is usually performed prior to feeding the input to the neural network, though methods for combining dimension reduction with neural networks has shown to be competitive[18].

2.4.7 3D Convolution

In a Convolutional Neural Network designed for hyperspectral images, a 3D convolution layer can utilise the spectral-spatial characteristics of the input data[19] by convolving a 3D kernel with the hyperspectral input data. These layers see the network learn features from the hyperspectral inputs autonomously, in contrast to the more manual traditional process of dimensionality reduction[20]. CNN architectures incorporating 3D convolution layers have demonstrated performance superior to other state-of-the-art networks using dimension reduction[19].

2.5 FPGA Acceleration

An Field-Programmable Gate Array (FPGA) is an Integrated Circuit (IC) equipped with Configurable Logic Blocks (CLBs) and other features that can be programmed by the user. "Field-Programmable" refers to the capability of being reconfigured after being manufactured. This uniquely separates FPGAs from fixed-hardware components like CPUs, GPUs, and DPSs by being highly customisable. This ability grants the FPGA superior performance[21] and/or greater efficiency[22] compared to its aforementioned fixed-hardware counterparts. An Application-Specific Integrated Circuit (ASIC) is effectively the lowest level of implementation for a specific task, and will outperform an FPGA in practically every measurable metric[23]. These, however, are practically impossible to reconfigure after production, thus requiring extensive pre-manufacturing testing.

2.5.1 FPGA Performance

FPGAs are shown to outperform CPUs and GPUs in compute intensive workloads. In [21], the authors obtain a 10x reduction in cycles required to compute Gaussian Elimination when comparing FPGA to CPU solutions. They also obtain a 3x reduction in cycles required compared to the GPU implementation. While this does not account for the clock periods difference between the technologies, it highlights the architectural advantages of having highly customisable hardware.

Comparing performance and energy efficiency of an FPGA implementation of vision kernels to an ASIC implementation, the FPGA falls short [23]. Their results show an average 8.7x increase in area and a 2.8 to 6.3 times reduction in raw computational performance when comparing their FPGA to ASIC implementations of this specific workload. Their performance and area results are shown in Figure 7.

2.5.2 Hardware-software co-design

Hardware-software co-design is a design methodology that enables the optimisation of system performance by jointly designing hardware and software components. In the context of FPGAs, hardware-software co-design can be leveraged to accelerate workloads by partitioning the computation between custom hardware circuits implemented on the FPGA fabric and software running on an embedded processor or an external host processor. This approach allows the design to exploit the parallelism and flexibility of FPGAs to implement application-specific hardware accelerators that can significantly improve the performance of compute-intensive tasks. By carefully partitioning the computation and optimising data transfer between hardware and software components, designers can achieve a high degree of performance and energy efficiency for accelerated workloads.

FPGAs such as the Zynq-7030 on board the HYPSO-1 has a host processor and a reconfigurable FPGA. The host processor typically handle general operational tasks and leaving computationally heavy tasks to be performed on the FPGA. The host machine is referred to as the Processing

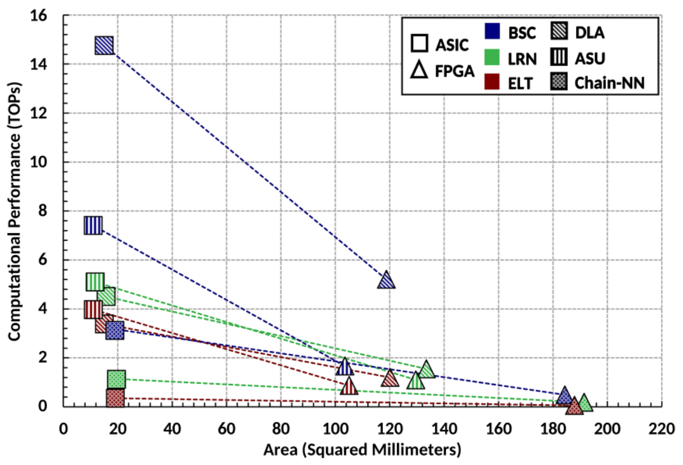


Figure 7: Area and performance gaps between FPGA and ASIC implementations of vision kernels. Obtained from [23].

System (PS), while the FPGA is referred to as the Programmable Logic (PL) or kernel. To avoid confusion with the kernel related to the convolution operation, the FPGA will be referred to as a *hardware kernel*.

2.6 High-Level Synthesis

High-Level Synthesis (HLS) is a process that enables designers to create hardware designs at a high level of abstraction, typically in high-level languages such as C or C++. It achieves this by translating the behavioural specification of the high-level code into appropriate Register-Transfer Level (RTL) structures, which can then be verified and refined using traditional design tools. The main benefits of using HLS include increased productivity, faster time-to-market, improved design quality and easier design optimisation [24].

While HLS usually enables rapid and comparatively easy development, the design process is not without its faults or shortcomings. Multiple industry standard HLS tools have been shown to provide erroneous results or fail to produce designs for valid programs outright [25]. While the performance of hardware developed using HLS are shown to greatly benefit from the speed-ups associated with running on ASIC circuits or FPGA platforms, it will not necessarily be comparable in performance to a highly optimised design developed in traditional RTL languages [26].

2.6.1 Vitis HLS

Vitis HLS is Xilinx' high-level synthesis tool, which translates C/C++ kernel code to RTL structures using their v++-compiler. It is a part of the Vitis unified software platform, which also includes the tools necessary for development of hardware accelerated applications. The Vitis tools allow for portability and reusability of such applications, by offloading the hardware-specific details from the design code. While a traditional accelerated application work flow is targeted at a single fixed platform, this approach allows portability between multiple devices [27].

2.6.2 Pragma directives

Pragmas are used in the C/C++ source code of the accelerated applications to apply certain optimisation techniques upon compiling and synthesising the selected hardware structures. They enable optimisations without changing the source code itself. Another technique to customise the synthesis process in Xilinx' Vitis tools is to specify optimisation directives. These are implemented in separate .tcl-files, and apply optimisations to their specified target solutions for quick design exploration. Certain pragmas are particularly useful in the development of CNNs on accelerated Xilinx platforms are listed below. Note that this list is not exhaustive. Documentation of HLS pragmas can be found in the Vitis HLS User Guide[24].

- HLS `stream`: Used to specify that a variable should be implemented as a FIFO. This pragma can be used to define the depth of the FIFO and its implementation type.
- HLS `inline`: Used to specify that a function should be inlined. It is useful to define whether a function should always be inlined, never be inlined, or whether it should be left up to the tool to decide.
- HLS `pipeline`: Used to specify that a loop should be pipelined. Defines the initiation interval of the pipeline and whether or not it should be rewound.
- HLS `dataflow`: Enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design. When the `dataflow` pragma is specified, Vivado HLS analyzes the dataflow between sequential functions or loops and creates channels (based on pingpong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed.
- HLS `array_partition`: Used to partition an array into individual elements or smaller arrays. Has the benefits of using RTL with multiple smaller memories and increasing the effective read and write ports for storage allowing potentially higher throughput, but requires more memory instances or registers.

3 Implementation

The implementation chapter presents the implemented convolution layer for semantic image segmentation of hyperspectral images. The use of High-Level Synthesis is motivated, and an overview of the convolution layer is presented.

3.1 Semantic Image Segmentation of Hyperspectral Images

The HYPSO mission aims to perform semantic image segmentation on its hyperspectral images. The hardware model motivated in [28] is used as a starting point. The proposed model was selected specifically for the HYPSO model and is based on a UNet network architecture, first proposed in [29].

3.1.1 UNet CNN

The architecture of the proposed UNet Convolutional Neural Network is shown in Figure 8. It includes 2D convolution, max-pooling, and 2D transposed convolution layers, with skip connections. The naming of the layers follows a convention of letters associated with the type of layer ('c' for convolution, 'p' for max-pooling, and 'u' for transposed convolution) with their positioning indicated by two numbers incrementing from left-to-right in the figure, which corresponds to how deep within the network they reside.

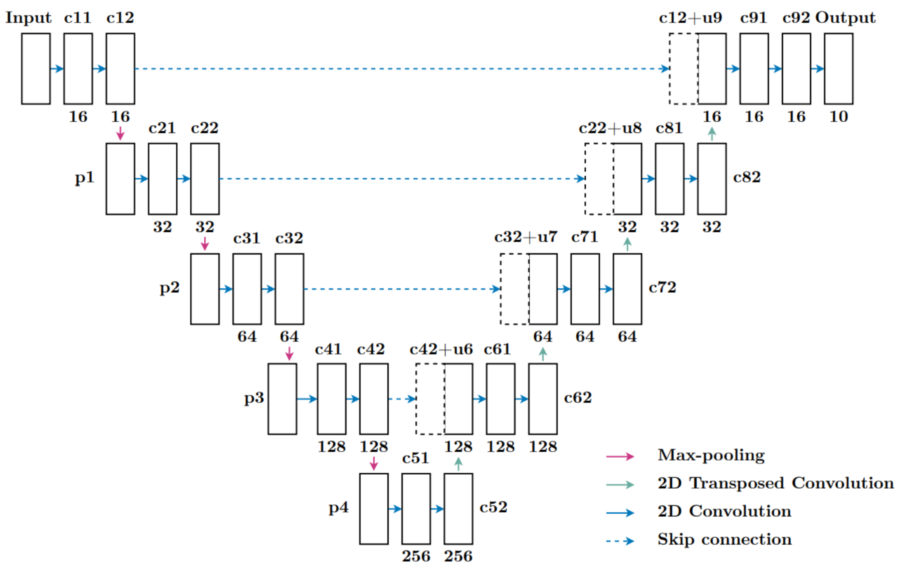


Figure 8: A descriptive model of the proposed UNet from [28]. Each layer is labelled with a name and has arrows representing the network operations.

Two slightly different models are proposed in [28]. They are based on the same UNet architecture, but differs in the bit-depth of the network parameters. The initial model, based on 32-bit weights and biases, will be referred to as *UNet*, while the quantised model using 16-bit weights and biases is referred to as the *quantised UNet*. The non-quantised is shown to have higher accuracy, and the 32-bit parameter size is used for implementation.

For the purposes of a hardware implementation, verifying the correctness of operation means the training of the network can be done separately. Thus, training is completely left out of this work. The results obtained in [28] could be used to showcase a full network demonstration by using the trained parameters. For testing and verification, however, arbitrary values can be used as long as they don't violate design constraints (i.e. by having large kernel and input values yielding products larger than what a 32-but integer can hold).

3.2 Convolution Layer

The convolution layers of the UNet will be prioritised for implementation, as their operations are shown to constitute a vast majority of total network computations[30][31]. Several techniques are used to improve throughput and resource utilisation of the convolution layer, such as a line buffer to reduce memory requirements of holding the input data, and extensive pipelining to minimise (among other things) the performance penalty of continuously fetching data from memory.

3.2.1 Requirements

The convolution layer is required to perform the convolution operation explained in Sections 2.4.1 on the fabric of the FPGA. A throughput proportional to one spatial pixel outputted to the feature map for each spectral band per clock cycle is desirable. Since the spectral data is mathematically uncorrelated in the convolutional layers of the UNet, i.e. the result of one spectral band does not depend on another, they can be processed in parallel. This equates to a performance requirement of $W \times H + T_{\text{setup}}(H, K)$ clock cycles to compute the convolution between an arbitrary kernel and the input image of W width and H height. $T_{\text{setup}}(H, K)$ is the setup time for the convolution hardware kernel. If the time required exhibits a limiting behaviour proportional to $W \times H$ with growing dimension values W and H, the setup time $T_{\text{setup}}(H, K)$ will diminish in relation to the total cycle cost, causing the performance requirement to be met. The convolution layer should accept inputs of up to 512×512 pixels in the spacial dimensions. The input and kernel weights must be imported from the PS into the PL, after which the result is stored for later use. The bias and activation functions can be added to the layer, but the performance and area requirements of these are insignificant to the convolution layer itself, so they can be omitted until multiple layers of the network is implemented simultaneously. Enabling the use of different strides is preferable, but not a hard requirement. Having a stride would only decrease the total workload, and adding it at a later stage should not contribute significantly toward area requirements. The UNet uses zero-padding and a kernel size of 3×3 and these configurations should be implemented.

3.2.2 Layer Design

The PL-implementation of the convolution layer is based on a load-compute-store pipeline that handles one convolution between the kernel and a window of the input image at the same time. The hardware kernel begins by loading the inputs, namely the image and kernel weights. It then creates a 3×3 window of the input image, which is zero-padded if required, which is passed to the block responsible for convolution. The matrices are convolved, and the output data is stored back to memory.

As one pixel of the output feature map is computed and stored, the next is already in the pipeline. The load function will import new pixels of the input image while the windowing function prepares a new 3×3 window at the same time as the convolution block convolves. Efficient pipelining should enable the hardware kernel to output one spatial pixel of information per cycle.

3.2.3 Line Buffer

A line buffer is used to create the sliding window that selects the 3×3 set of data to be convolved. A naive approach might load the entire input image into PL, as shown in Figure 9. This would lead to unnecessary use of logic area, as the majority of the data would be sitting idle while irrelevant to the current calculation, or already be used completely. Instead, the line buffer is used to hold only the input values in close temporally proximity to the target pixel. The input image is sequentially loaded in the BIL-format explained in Section 2.2.2. For the explanation of the line buffer, assume a spectral depth of one, which effectively makes the input a 2D image. The hardware kernel must load an entire line of horizontal data before the next column becomes available. Two full lines plus three additional cells of the input image is stored at a time in the line buffer, which is visualised in Figure 10. Importing a new pixel at the beginning of the line buffer and shifting the other values by one every cycle will automatically create a new window. When a pixel reaches the end of the line buffer it will no longer be required and is discarded. The blue 3×3 square in Figure 10 indicates the window, while the labels A and B signify that the pixels flow over to the next row. The values are shift to the left in the figure, as indicated by the arrow.

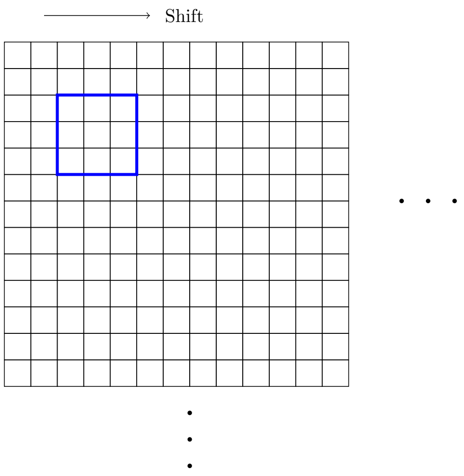


Figure 9: Visualisation of a naive implementation of the moving kernel window.

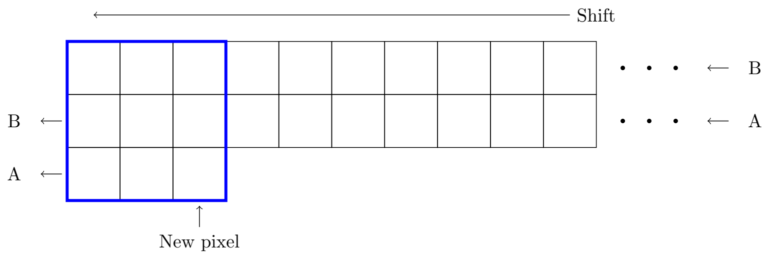


Figure 10: Line buffer visualisation.

An additional buffering technique is also implemented. This technique is similar in design to the line buffer, but avoids shifting every element in the long line buffer array. To differentiate between the buffering techniques this new technique is referred to as a *column select* line buffer, or simply *colselect*. It uses the same principle of shifting one new pixel in and discarding one old pixel every cycle, but instead of shifting the entire array it selects a column of its $W_i \times (W_k - 1)$ array and updates only those elements every cycle. The window holding the currently active pixels is shifted every cycle, with one new pixel imported from the input stream, and the two other being imported from the selected column. When pixels get shifted out of the 3×3 window they get discarded. These pixels do not need to be re-stored because the buffering array is wide enough to hold two entire lines of the input image. A visualisation of the colselect buffer is shown in Figure 11.

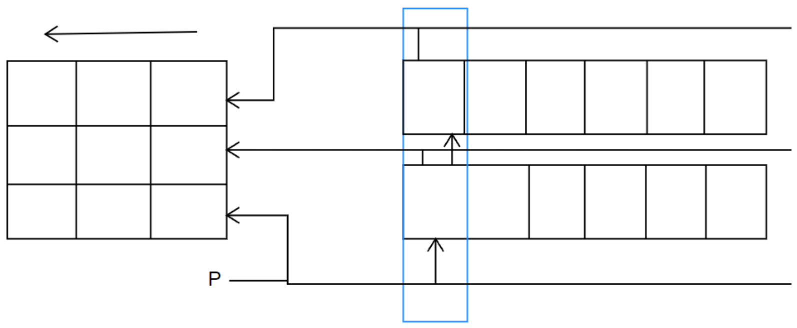


Figure 11: Visualisation of the colselect buffering technique.

The blue rectangle indicate the currently selected column and the arrows indicate the flow of data.

The column selector slides one column across every clock cycle, eventually wrapping around. When the selector moves, the arrows of the figure move with it, effetcively enabling and disabling one row at a time. A new pixel is imported every cycle, labelled in the figure as 'P'. The cells in the figure are not electrically transparent. In other words, data does not propagate though multiple arrows in on cycle, instead replacing the old data that was there after it has propagated. They can be thought of as implemented with D flip flops.

The colselect buffer design should allow the HLS tool to optimise the design more than the line buffer approach. This is because the line buffer has to update every value in its array every cycle, which forces more operations on a larger set of data. The colselect buffer only needs access to its 3×3 window and two storage cells every cycle, meaning a clever tool could store the temporarily inactive values in the FPGA's block RAM or other deep storage until they are needed.

Both buffering schemes have an Initiation Interval (II) equivalent to the input image width plus two. This is because both buffers need to fill their windows before they can be served to the convolution block. One line of data plus two more pixels is also the lowest possible number of inputs one needs to read to start convolution of images served in row-major order for a 3×3 kernel.

3.2.4 Handling Higher Dimensionality

So far the convolution hardware kernel has only concerned itself with 2D inputs. A mechanism to enable multidimensional convolution is to distribute the multidimensional input to many parallel execution cores. As explained in Section 3.2.1, there is no dependence between spectral bands in the convolution layer, and they can be computed separately. Thus, a mechanism for distributing the input data across multiple convolution cores is used.

The block that loads the input data is responsible for distributing the input data to the correct spectral core. The BIL format encodes one line of one spectral band at a time, before moving on to the next spectral dimension. After all the spectral information contained in the first line is encoded, the next spatial line is up. The block that loads data from memory knows the spatial dimensions of the input image W and H, and will serve the first W values to one core's buffer before it starts serving the next. This way every spectral dimension receives one line of data before the first spectral band starts receiving its second line where it can start computing the convolution.

Storing the data is performed in a similar way. The outputs from the convolution cores are interleaved by spectral band and stored back to the PS. The cores should have constant execution cycle time and each core starts sequentially when they receive enough inputs from the input block. Thus, the storage block should always receive data in the correct BIL-formatted order.

3.2.5 HLS Implementation

HLS is used for implementation over Hardware Definition Language (HDL) because of its relative ease of development. Vitis HLS is chosen as the platform as it is compatible with the target FPGA, and the chosen programming language is C++. The target clock frequency set for the Vitis tool is 150MHz, but this can be adjusted depending on the latency of the kernel's critical path.

The hardware kernel source code starts with several `define` statements, shown in Listing 1. The function of most of these should be self-explanatory. `pixel_t` is the type used for practically all numbers in the dataflow and is set to a 32-bit integer. `NUM_CHANNELS` defines the spectral depth of the input data, and this must match the spectral dimension set in the host code. `LINE_BUFFER_COSELECT` is defined such that a conditional compilation can select between the two buffering techniques described in Section 3.2.3. The different implementations can be switched between by defining either `LINE_BUFFER_COLSELECT` or `LINE_BUFFER_SHIFTREG`.

```
#define KERNEL_WIDTH 3
#define KERNEL_HEIGHT 3
#define KERNEL_SIZE (KERNEL_WIDTH * KERNEL_HEIGHT)
#define pixel_t int32_t
```

```
#define MAX_IMG_WIDTH 512
#define MAX_IMG_HEIGHT 512
#define MAX_IMG_SIZE (MAX_IMG_WIDTH*MAX_IMG_HEIGHT)
#define NUM_CHANNELS 10 // Must match NUM_CHANNELS in host code
#define LINE_BUFFER_COLSELECT // Line buffer type selector: "COLSELECT" or "
    SHIFTREG"
```

Listing 1: Source code #define statements

The dataflow of the hardware kernel becomes apparent by looking at the ”main” function of the source code shown in Listing 2. The entire main function has to be defined as `extern "C"` to avoid name mangling issues within Vitis. The arguments to the main function are explained in the code listing. The `HLS INTERFACE` pragma is used to specify how RTL ports are infered from the function arguments during synthesis. It is only supported in the top-level function.

The hardware kernel source code passes around data using the `hls::stream` object. It synthesises into FIFOs, and can be viewed as data channels between functions. `hls::stream` accepts most C++ types, including user-defined types, with the exception of user-defined classes and structures that contain member functions. The streams used in the main function are defined in arrays of streams. This is because each spectral dimension of the input uses its own core with a unique stream.

```
extern "C" {

/*
    Convolution Kernel

    Arguments:
        in1    (input)  --> Input kernel coefficients
        in2    (input)  --> Input image
        out    (output) --> Output image
        width  (input)  --> Width of input image
        height(input)   --> Height of input image
*/

void conv(pixel_t* in1, pixel_t* in2, pixel_t* out, int width, int height ) {

#pragma HLS INTERFACE m_axi port = in1 bundle = gmem0
#pragma HLS INTERFACE m_axi port = in2 bundle = gmem1
#pragma HLS INTERFACE m_axi port = out bundle = gmem1
#pragma HLS INTERFACE s_axilite port=width
#pragma HLS INTERFACE s_axilite port=height

  static hls::stream<conv_window_t> window_stream[NUM_CHANNELS];
  static hls::stream<pixel_t> krnl_stream[NUM_CHANNELS];
  static hls::stream<pixel_t> img_stream[NUM_CHANNELS];
  static hls::stream<pixel_t> out_stream[NUM_CHANNELS];

#pragma HLS DATAFLOW
  load_inputs(in1, krnl_stream, in2, img_stream, width, height);
  multidimensional_conv<NUM_CHANNELS>(window_stream, krnl_stream, img_stream,
    out_stream, width, height);
  store_result(out, out_stream, width, height);
}
}
```

Listing 2: Source code main function

The dataflow of the hardware kernel is as follows:

1. `load_inputs()` loads the input image and kernel weights and produces data for streams to be consumed by the `create_conv_window()` and `compute_conv()` functions.
2. `multidimensional_conv()` initiates the spectral cores. It calls `create_conv_window()` and `compute_conv()` for every channel and passes the correct streams as arguments.
3. `create_conv_window()` uses a line buffer to create a window that is passed to `compute_conv()` for convolution.

4. `compute_conv()` initialises by reading the kernel weights from its associated `kernel_stream`. After initialisation the function performs convolution with the kernel and window received from `create_conv_window()`. The result is passed to `store_results()`.

5. `store_results()` saves the computed output data back to memory.

The load/compute/store coding style is used for the HLS hardware kernel source code. The load and store operation handle memory access, and are designed to move data in and out of the hardware kernel efficiently. They are shown in Listings 3 and 4 respectively.

```
static void load_inputs(
        pixel_t* kernel_src,
        hls::stream<pixel_t> kernel_stream[NUM_CHANNELS],
        pixel_t* image_src,
        hls::stream<pixel_t> image_stream[NUM_CHANNELS],
        int image_width,
        int image_height)
{

  int i=0;
kernel_load:
  for(int y=0; y<KERNEL_HEIGHT; y++) {
    for(int c=0; c<NUM_CHANNELS; c++) {
      for(int x=0; x<KERNEL_WIDTH; x++) {
#pragma HLS LOOP_TRIPCOUNT max=KERNEL_SIZE*NUM_CHANNELS
        kernel_stream[c] << kernel_src[i];
        i++;
      }
    }
  }

  int j=0;
image_load:
for(int y=0; y<image_height; y++) {
    for(int c=0; c<NUM_CHANNELS; c++) {
      for(int x=0; x<image_width; x++) {
#pragma HLS LOOP_TRIPCOUNT max=MAX_IMG_SIZE*NUM_CHANNELS
        image_stream[c] << image_src[j];
        j++;
      }
    }
  }
}
```

Listing 3: load_inputs() function

```
static void store_result(
        pixel_t* out,
        hls::stream<pixel_t> output_stream[NUM_CHANNELS],
        int image_width,
        int image_height)
{
  int i=0;
output_store:
  for(int y=0; y<image_height; y++) {
      for(int c=0; c<NUM_CHANNELS; c++) {
        for(int x=0; x<image_width; x++) {
#pragma HLS LOOP_TRIPCOUNT max=MAX_IMG_SIZE*NUM_CHANNELS
          out[i] = output_stream[c].read();
          i++;
        }
      }
  }
}
```

Listing 4: load_inputs() function

The load and store function in Listing 3 and 4 use for-loops to iterate over the input and output data as described in Section 3.2.4. The inclusion of `pragma HLS LOOP_TRIPCOUNT` is for analysis only, and does not impact synthesis. They report the tripcount to the analysis tool.

The function responsible for implementing the line buffers is `create_conv_window()`. There are two implementations for this function. The line buffer implementation is shown in Listing 5 and the colselect implementation is shown in Listing 6.

```
static void create_conv_window(
            hls::stream<pixel_t>& image_stream,
            hls::stream<conv_window_t>& window_stream,
            int image_width,
            int image_height) {
  // Shift register requires 2 rows of image plus 1 row of kernel
  const int line_buffer_size = MAX_IMG_WIDTH*2 + KERNEL_WIDTH;
  // Limit size for when image width < MAX WIDTH (hw can't be dynamically allocated
    )
  const int line_buffer_actual_size = image_width*2 + KERNEL_WIDTH;
  // Create line buffer
  pixel_t line_buffer[line_buffer_size];
  // Pragmas for access partitioning and loop dependencies
#pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=0

  // Create sliding window matching the kernel dimensions
  conv_window_t Window;

  int init_iterations = (KERNEL_WIDTH-1)/2 + image_width;
  int num_pixels = image_width*image_height;
  int total_iterations = init_iterations + num_pixels;

  const int max_iterations = (KERNEL_WIDTH-1)/2 + MAX_IMG_WIDTH + MAX_IMG_SIZE;

update_conv_window_1:
  for(int n=0; n<total_iterations; n++) {
#pragma HLS LOOP_TRIPCOUNT max=total_iterations
#pragma HLS PIPELINE II=1

    pixel_t new_pixel = (n<num_pixels) ? image_stream.read() : 0;

    // Shift line buffer array
    for(int i=0; i<line_buffer_actual_size-1; i++) {
      line_buffer[i] = line_buffer[i+1];
    }
    line_buffer[line_buffer_actual_size-1] = new_pixel;

    // Create window
read_window_from_lb:
    for(int y=0; y<KERNEL_HEIGHT; y++){
      for(int x=0; x<KERNEL_WIDTH; x++) {
        Window.pixel[y][x] = line_buffer[y * image_width + x];
      }
    }

    // Write out the updated window to stream if finished initializing
    if (n >= init_iterations) window_stream.write(Window);
  }
}
```

Listing 5: Line buffer implementation of the create_conv_window() function

```
static void create_conv_window(
        hls::stream<pixel_t>& image_stream,
        hls::stream<conv_window_t>& window_stream,
        int image_width,
        int image_height)
{
  // Create line buffer
  pixel_t line_buffer[KERNEL_HEIGHT-1][MAX_IMG_WIDTH];
  // Pragmas for access partitioning and loop dependencies
#pragma HLS ARRAY_PARTITION variable=line_buffer dim=1 complete
#pragma HLS DEPENDENCE variable=line_buffer inter false
#pragma HLS DEPENDENCE variable=line_buffer intra false

  // Create sliding window matching the kernel dimensions
  conv_window_t Window;

  // int to track coloumn of sliding window
  int col = 0;
  // Number of additional iterations to populate line and window buffers
  int init_iterations = (KERNEL_WIDTH-1)/2 + image_width*((KERNEL_WIDTH-1)/2 + (
    KERNEL_HEIGHT-1)/2)/2;
  int num_pixels = image_width*image_height;
  int total_iterations = num_pixels + init_iterations;

  const int MAX_ITERATIONS = MAX_IMG_WIDTH*MAX_IMG_HEIGHT + MAX_IMG_WIDTH*((
    KERNEL_WIDTH-1)/2+(KERNEL_HEIGHT-1)/2)/2;

update_conv_window_0:
  for (int n=0; n<total_iterations; n++) {
#pragma HLS LOOP_TRIPCOUNT max=MAX_ITERATIONS
#pragma HLS PIPELINE II=1

    // Read new pixel from image
    pixel_t next_pixel = (n<num_pixels) ? image_stream.read() : 0;

    // Shift window
shift_window:
    for (int i = 0; i < KERNEL_HEIGHT; i++) {
      for (int j = 0; j < KERNEL_WIDTH-1; j++) {
        Window.pixel[i][j] = Window.pixel[i][j+1];
      }
      // At final coloumn get new values from line_buffer or the next pixel
      Window.pixel[i][KERNEL_WIDTH-1] = (i < KERNEL_HEIGHT-1) ? line_buffer[i][col]
     : next_pixel;
    }
shift_linebuffer_col:
    // Shift line_buffer and add new pixel
    for (int i = 0; i < KERNEL_HEIGHT-2; i++) {
      line_buffer[i][col] = line_buffer[i+1][col];
    }
    line_buffer[KERNEL_HEIGHT-2][col] = next_pixel;

    // Update coloumn tracker
    if (col == (image_width-1)) {
      col = 0;
    } else {
      col++;
    }

    // Write out the updated window to stream if finished initializing
    if (n >= init_iterations) window_stream.write(Window);
  }
}
```

Listing 6: Colselect implementation of the create_conv_window() function

The function responsible for computing the convolution between the kernel and image window is `compute_conv()`. It is shown in Listing 7. It reads the kernel weights from `load_inputs()` and start the main loop called `full_image_conv` (the name refers to a full 2D image, not the entire cube for hyperspectral inputs). This function is also responsible for zero-padding when the convolution being performed is at the image border.

```
static void compute_conv(
        hls::stream<pixel_t>& output_stream,
        hls::stream<pixel_t>& kernel_stream,
        hls::stream<conv_window_t>& window_stream,
        int image_width,
        int image_height)
{
  pixel_t kernel[KERNEL_HEIGHT][KERNEL_WIDTH];
#pragma HLS ARRAY_PARTITION variable=kernel complete dim=0

  // Load kernel coefficients
load_coeffs_to_compute:
  for (int i = 0; i < KERNEL_HEIGHT; i++) {
    for (int j = 0; j < KERNEL_WIDTH; j++) {
#pragma HLS LOOP_FLATTEN
#pragma HLS PIPELINE II=1
```

```
      kernel[i][j] = kernel_stream.read();
    }
  }

  // Compute convolution
full_image_conv:
  for (int y = 0; y < image_height; y++) {
#pragma HLS LOOP_TRIPCOUNT max=MAX_IMG_HEIGHT
    for (int x = 0; x < image_width; x++) {
#pragma HLS LOOP_TRIPCOUNT max=MAX_IMG_WIDTH
#pragma HLS PIPELINE II=1
      conv_window_t Window = window_stream.read();

      // Compute cross-correlation between window and kernel
      pixel_t sum = 0;
pixel_conv:
      for(int row = 0; row < KERNEL_HEIGHT; row++) {
        for (int col=0; col < KERNEL_WIDTH; col++) {
          pixel_t new_pixel;

          int x_offset = (x+col-(KERNEL_WIDTH/2));
          int y_offset = (y+row-(KERNEL_HEIGHT/2));

          // Boundary conditions: Set to 0 outside image frame
          if ((x_offset<0) || (x_offset>=image_width) || (y_offset<0) || (y_offset
    >=image_height)) {
            new_pixel = 0;
          } else {
            new_pixel = Window.pixel[row][col];
          }
          sum += new_pixel * kernel[row][col];
        }
      }

      output_stream.write(sum);
    }
  }
}
```

Listing 7: Colselect implementation of the create_conv_window() function

3.2.6 Performance Estimation

The ideal performance of the convolutional layer can be estimated from the requirements. The spatial dimensions of the input image are 512 × 512 which equals 262 144 pixels. Throughput of one pixel per clock cycle and a target clock frequency of 150MHZ yields the following runtime:

$$\frac{\text{Total pixels}}{1\text{px/cycle} \times 150MHz} = \frac{262144}{150}\mu s \approx 1.75ms$$

This includes convolution of the entire hyperspectral input, as the layers are processed in parallel. It does not include the overhead of initialising the hardware kernel. As discussed earlier, some blocks need to initialise internal structures before they can start pipelined operation. As mentioned in Section 3.2.3, the input buffering techniques require at least 514 cycles to initialise for a 2D image. This, however only amount to a minuscule $512/262144 \approx 0.2\%$ of total estimated cycles. The computation of the deepest spectral band will begin last, and will have a much greater Initiation Interval. Using 10 channels, it will have to wait for $512 \times 19 + 2 = 9730$ cycles before it can start convolution. The 19 comes from waiting for the input to read 19 spatial lines before it receives data from its second spatial line. This II amounts to $9730/262144 \approx 3.7\%$ of the total cycles and bumps the estimated completion time to 1.81ms.

3.2.7 Evaluation

Evaluation of the convolution layer is performed by comparing the estimated performance in Section 3.2.6 to the numbers reported by the Vitis tools. Vitis is able to run software simulations and hardware emulations of the design. Software simulations are useful for quickly getting an indication of design correctness, but can have inaccuracies. Hardware emulation use QEMU for the host and RTL and System-C-based emulation to co-simulate the design and provide a complete execution model. Upon building the design for hardware emulation various reports are generated. These can be inspected to see resource usage and timing estimates for the design.

The two line buffer implementations will be synthesised and compared using a $512 \times 512 \times 1$ dimension configuration. The line buffers are duplicated for extra dimensions, so having one dimension should yield similar results to a configuration of high spectral depth.

The design is synthesised for various dimension configurations. The maximum input image dimensions as well as the spectral depth is configurable. A set of configuration will be synthesised with the buffering technique that shows the best results:

1. 2D: A $512 \times 512 \times 1$ configuration. Shows the resource usage when the design is synthesised to a regular 2D convolution layer.
2. Minimum size: A $64 \times 64 \times 1$ configuration. Shows the resource utilisation for a network configured for very small, monochromatic image inputs.
3. Maximum size: A $512 \times 512 \times D$ configuration. D is the highest depth where the FPGA can still fit an entire convolution layer, and is found though incrementally increasing D until the layer no longer fits. Shows how deep of an image the design can handle while retaining a large spatial resolution of 512×512.
4. Realistic: A $512 \times 512 \times 10$ configuration. Shows the resource utilisation and performance for a realistic convolution layer where some dimension reduction has been performed on the hyperspectral images to reduce it to 10 spectral dimensions.

3.3 Full Network

Multiple types of network layers were not implemented. Future work could add bias and activation functions to the convolution layer, as well as implementing the other network layers mentioned in Section 2.4. A fully realised CNN can then be run on the Zynq platform.

It is unrealistic to expect that an entire CNN the size of UNet can fit inside the FPGA. A workaround for this is to implement different layer types in the FPGA fabric and scheduling them to run from from PS when required. This would require significant memory bandwidth, but it could be a realistic way to implement arbitrarily deep neural networks (as long as data stored by skip connections don't surpass the memory constraints of the platform).

4 Results and Discussion

This chapter presents and discusses the result from the convolution layer implementation as well as discussing possible improvements and future work. The results from synthesising various layer configurations are presented first, followed by validating the performance of the convolution layer. The implications of these are discussed, before discussion on future work and possible improvements conclude the chapter

4.1 Line Buffer Results

The two buffering techniques, line buffer and colselect were synthesised for a $512 \times 512 \times 1$ design configuration. The reports generated after synthesis are shown in Figures 12 and 13.

The reports estimate that the latency of the techniques are very similar, only differing by 359 cycles amounting to 0.03ms of latency, but they differ wildly in resource utilisation. The line buffer implementation used 37711 flip-flops, a full 8% for a relatively small layer configuration of only one spectral dimension. Meanwhile, the colselect implementation utilises 5217 flip-flops and two BRAM blocks (out of the total 312 blocks on the target FPGA). The colselect scheme uses slightly more LUTs as well. Because of these results, the colselect is used for the remaining tests. The reason for this discrepancy is most likely for the reasons explained in Section 3.2.3.

4.2 Various Configuration of the Convolution Layer

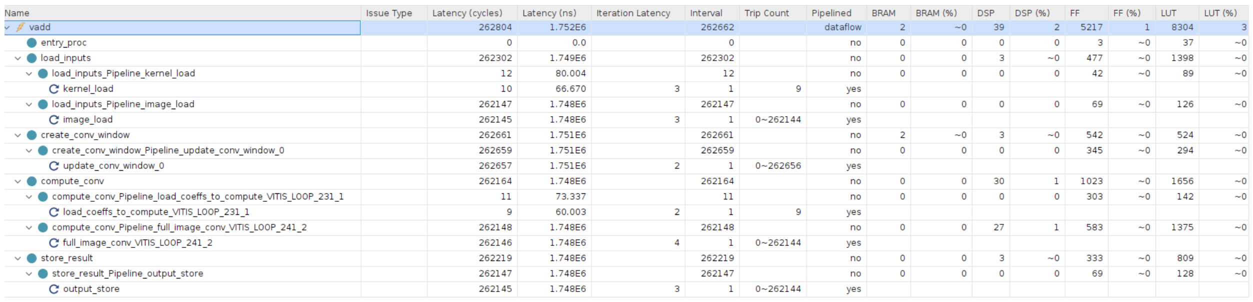
The results from synthesising the various system configurations listed Section 3.2.7 in are shown in Figures 141516171819.

For maximum depth, a value of 70 was reached before the tool crashed upon synthesis. At this point, DSP utilisation is above 100%. This is obviously not satisfactory, but a depth of 64 layers was synthesised and confirmed to fit on the FPGA.

The minimum configuration had a very low resource utilisation, as expected. The 2D-configuration represents single core performance and utilisation.

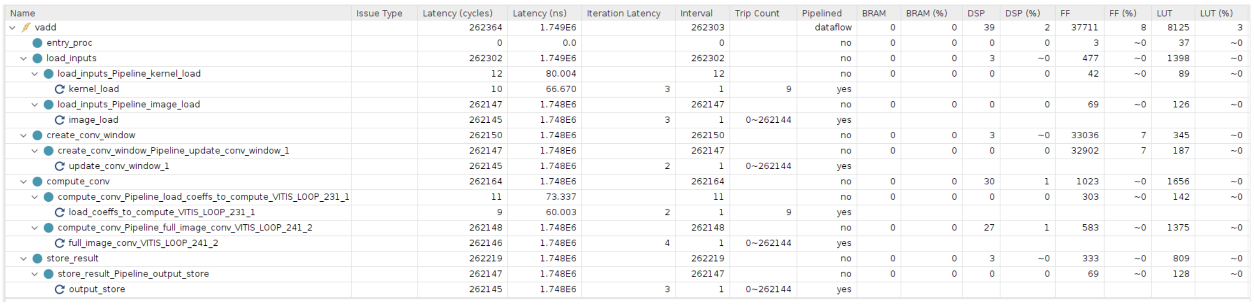
It can be observed from all the reports that the worst (absolute) latency was just over 1.75ms. This represents the run time for convolving an image of spatial resolution 512×512.

Time did unfortunately not allow a deep discussion beyond what has been noted earlier in the thesis.



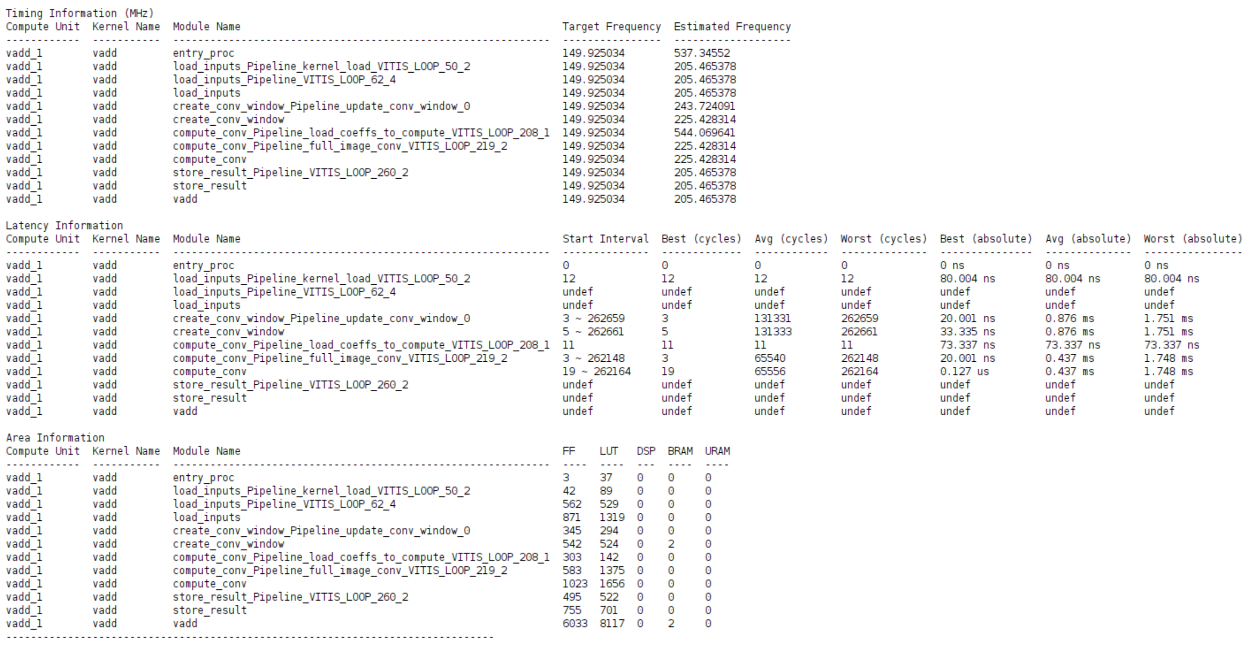
Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
vadd		262804	1.752E6		262662		dataflow	2	~0	39	2	5217	1	8304	3
entry_proc		0	0.0		0		no	0	0	0	0	3	~0	37	~0
load_inputs		262302	1.749E6		262302		no	0	0	3	~0	477	~0	1398	~0
load_inputs_Pipeline_kernel_load		12	80.004		12		no	0	0	0	0	42	~0	89	~0
kernel_load		10	66.670	3	1	9	yes								
load_inputs_Pipeline_image_load		262147	1.748E6		262147		no	0	0	0	0	69	~0	126	~0
image_load		262145	1.748E6	3	1	0~262144	yes								
create_conv_window		262661	1.751E6		262661		no	2	~0	3	~0	542	~0	524	~0
create_conv_window_Pipeline_update_conv_window_0		262659	1.751E6		262659		no	0	0	0	0	345	~0	294	~0
update_conv_window_0		262657	1.751E6	2	1	0~262656	yes								
compute_conv		262164	1.748E6		262164		no	0	0	30	1	1023	~0	1656	~0
compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_231_1		11	73.337		11		no	0	0	0	0	303	~0	142	~0
load_coeffs_to_compute_VITIS_LOOP_231_1		9	60.003	2	1	9	yes								
compute_conv_Pipeline_full_image_conv_VITIS_LOOP_241_2		262148	1.748E6		262148		no	0	0	27	1	583	~0	1375	~0
full_image_conv_VITIS_LOOP_241_2		262146	1.748E6	4	1	0~262144	yes								
store_result		262219	1.748E6		262219		no	0	0	3	~0	333	~0	809	~0
store_result_Pipeline_output_store		262147	1.748E6		262147		no	0	0	0	0	69	~0	128	~0
output_store		262145	1.748E6	3	1	0~262144	yes								

Figure 12: Report from synthesising the convolution layer with the colselect buffering technique.



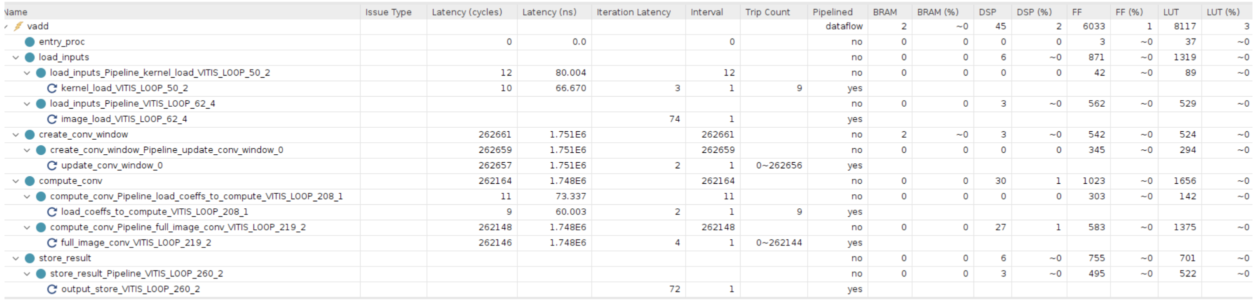
Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
vadd		262364	1.749E6		262303		dataflow	0	0	39	2	37711	8	8125	3
entry_proc		0	0.0		0		no	0	0	0	0	3	~0	37	~0
load_inputs		262302	1.749E6		262302		no	0	0	3	~0	477	~0	1398	~0
load_inputs_Pipeline_kernel_load		12	80.004		12		no	0	0	0	0	42	~0	89	~0
kernel_load		10	66.670	3	1	9	yes								
load_inputs_Pipeline_image_load		262147	1.748E6		262147		no	0	0	0	0	69	~0	126	~0
image_load		262145	1.748E6	3	1	0~262144	yes								
create_conv_window		262150	1.748E6		262150		no	0	0	3	~0	33036	7	345	~0
create_conv_window_Pipeline_update_conv_window_1		262147	1.748E6		262147		no	0	0	0	0	32902	7	187	~0
update_conv_window_1		262145	1.748E6	2	1	0~262144	yes								
compute_conv		262164	1.748E6		262164		no	0	0	30	1	1023	~0	1656	~0
compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_231_1		11	73.337		11		no	0	0	0	0	303	~0	142	~0
load_coeffs_to_compute_VITIS_LOOP_231_1		9	60.003	2	1	9	yes								
compute_conv_Pipeline_full_image_conv_VITIS_LOOP_241_2		262148	1.748E6		262148		no	0	0	27	1	583	~0	1375	~0
full_image_conv_VITIS_LOOP_241_2		262146	1.748E6	4	1	0~262144	yes								
store_result		262219	1.748E6		262219		no	0	0	3	~0	333	~0	809	~0
store_result_Pipeline_output_store		262147	1.748E6		262147		no	0	0	0	0	69	~0	128	~0
output_store		262145	1.748E6	3	1	0~262144	yes								

Figure 13: Report from synthesising the convolution layer with the line buffer buffering technique.



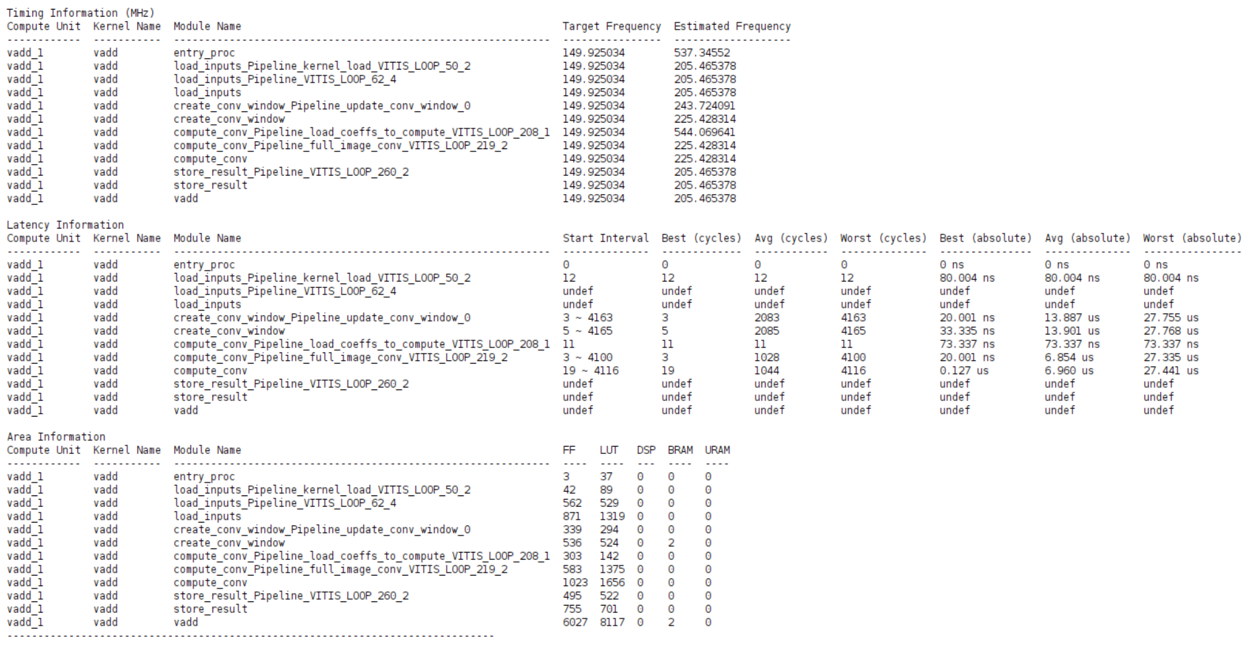
```
Timing Information (MHz)
Compute Unit  Kernel Name  Module Name                                                  Target Frequency  Estimated Frequency
------------  -----------  -----------------------------------------------------------  ----------------  -------------------
vadd_1        vadd         entry_proc                                                   149.925034        537.34552
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2             149.925034        205.465378
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                         149.925034        205.465378
vadd_1        vadd         load_inputs                                                  149.925034        205.465378
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0             149.925034        243.724091
vadd_1        vadd         create_conv_window                                           149.925034        225.428314
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1 149.925034        544.069641
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2       149.925034        225.428314
vadd_1        vadd         compute_conv                                                 149.925034        225.428314
vadd_1        vadd         store_result_Pipeline_VITIS_LOOP_260_2                       149.925034        205.465378
vadd_1        vadd         store_result                                                 149.925034        205.465378
vadd_1        vadd         vadd                                                         149.925034        205.465378

Latency Information
Compute Unit  Kernel Name  Module Name                                                  Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolute)  Worst (absolute)
------------  -----------  -----------------------------------------------------------  --------------  -------------  ------------  --------------  ---------------  --------------  ----------------
vadd_1        vadd         entry_proc                                                   0               0              0             0               0 ns             0 ns            0 ns
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2             12              12             12            12              80.004 ns        80.004 ns       80.004 ns
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                         undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         load_inputs                                                  undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0             3 ~ 262659      3              131331        262659          20.001 ns        0.876 ms        1.751 ms
vadd_1        vadd         create_conv_window                                           5 ~ 262661      5              131333        262661          33.335 ns        0.876 ms        1.751 ms
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1 11              11             11            11              73.337 ns        73.337 ns       73.337 ns
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2       3 ~ 262148      3              65540         262148          20.001 ns        0.437 ms        1.748 ms
vadd_1        vadd         compute_conv                                                 19 ~ 262164     19             65556         262164          0.127 us         0.437 ms        1.748 ms
vadd_1        vadd         store_result_Pipeline_VITIS_LOOP_260_2                       undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         store_result                                                 undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         vadd                                                         undef           undef          undef         undef           undef            undef           undef

Area Information
Compute Unit  Kernel Name  Module Name                                                  FF    LUT   DSP  BRAM  URAM
------------  -----------  -----------------------------------------------------------  ----  ----  ---  ----  ----
vadd_1        vadd         entry_proc                                                   3     37    0    0     0
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2             42    89    0    0     0
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                         562   529   0    0     0
vadd_1        vadd         load_inputs                                                  871   1319  0    0     0
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0             345   294   0    0     0
vadd_1        vadd         create_conv_window                                           542   524   0    2     0
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1 303   142   0    0     0
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2       583   1375  0    0     0
vadd_1        vadd         compute_conv                                                 1023  1656  0    0     0
vadd_1        vadd         store_result_Pipeline_VITIS_LOOP_260_2                       495   522   0    0     0
vadd_1        vadd         store_result                                                 755   701   0    0     0
vadd_1        vadd         vadd                                                         6033  8117  0    2     0
-----------------------------------------------------------------------------------
```

Figure 14: Synthesis report for synthesising the 2D design configuration.



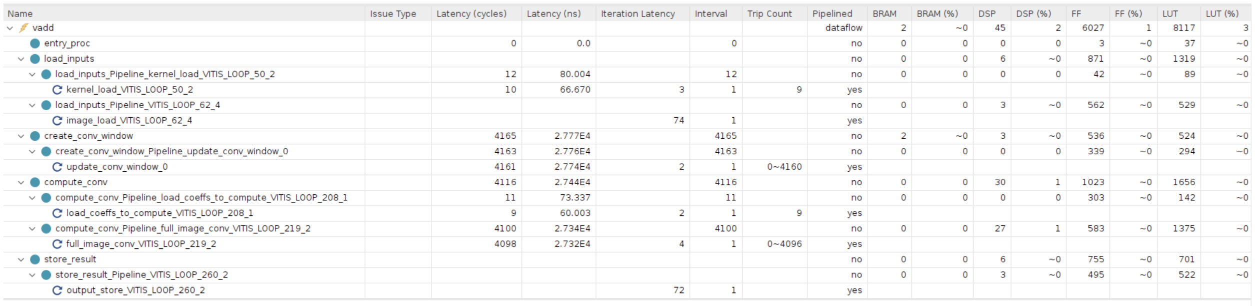
Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
vadd							dataflow	2	~0	45	2	6033	1	8117	3
entry_proc		0	0.0		0		no	0	0	0	0	3	~0	37	~0
load_inputs							no	0	0	6	~0	871	~0	1319	~0
load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2		12	80.004		12		no	0	0	0	0	42	~0	89	~0
kernel_load_VITIS_LOOP_50_2		10	66.670	3	1	9	yes								
load_inputs_Pipeline_VITIS_LOOP_62_4							no	0	0	3	~0	562	~0	529	~0
image_load_VITIS_LOOP_62_4				74	1		yes								
create_conv_window		262661	1.751E6		262661		no	2	~0	3	~0	542	~0	524	~0
create_conv_window_Pipeline_update_conv_window_0		262659	1.751E6		262659		no	0	0	0	0	345	~0	294	~0
update_conv_window_0		262657	1.751E6	2	1	0~262656	yes								
compute_conv		262164	1.748E6		262164		no	0	0	30	1	1023	~0	1656	~0
compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1		11	73.337		11		no	0	0	0	0	303	~0	142	~0
load_coeffs_to_compute_VITIS_LOOP_208_1		9	60.003	2	1	9	yes								
compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2		262148	1.748E6		262148		no	0	0	27	1	583	~0	1375	~0
full_image_conv_VITIS_LOOP_219_2		262146	1.748E6	4	1	0~262144	yes								
store_result							no	0	0	6	~0	755	~0	701	~0
store_result_Pipeline_VITIS_LOOP_260_2							no	0	0	3	~0	495	~0	522	~0
output_store_VITIS_LOOP_260_2				72	1		yes								

Figure 15: Synthesis report estimate for synthesising the 2D design configuration



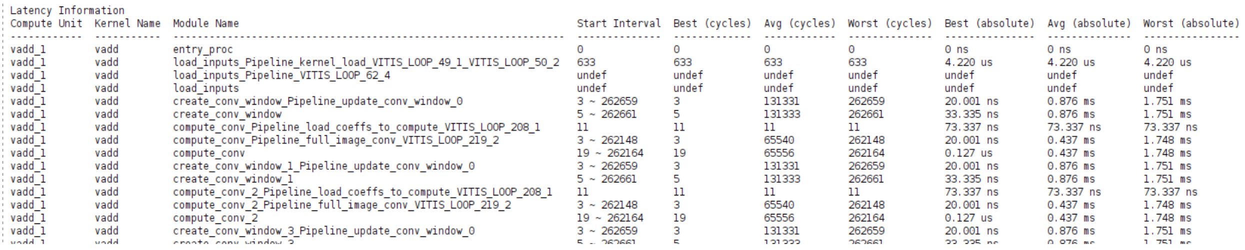
```
Timing Information (MHz)
Compute Unit  Kernel Name  Module Name                                                    Target Frequency  Estimated Frequency
------------  -----------  -------------------------------------------------------------  ----------------  -------------------
vadd_1        vadd         entry_proc                                                     149.925034        537.34552
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2               149.925034        205.465378
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                           149.925034        205.465378
vadd_1        vadd         load_inputs                                                    149.925034        205.465378
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0               149.925034        243.724091
vadd_1        vadd         create_conv_window                                             149.925034        225.428314
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1  149.925034        544.069641
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2         149.925034        225.428314
vadd_1        vadd         compute_conv                                                   149.925034        225.428314
vadd_1        vadd         store_result_Pipeline_VITIS_LOOP_260_2                         149.925034        205.465378
vadd_1        vadd         store_result                                                   149.925034        205.465378
vadd_1        vadd         vadd                                                           149.925034        205.465378

Latency Information
Compute Unit  Kernel Name  Module Name                                                    Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolute)  Worst (absolute)
------------  -----------  -------------------------------------------------------------  --------------  -------------  ------------  --------------  ---------------  --------------  ----------------
vadd_1        vadd         entry_proc                                                     0               0              0             0               0 ns             0 ns            0 ns
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2               12              12             12            12              80.004 ns        80.004 ns       80.004 ns
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                           undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         load_inputs                                                    undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0               3 ~ 4163        3              2083          4163            20.001 ns        13.887 us       27.755 us
vadd_1        vadd         create_conv_window                                             5 ~ 4165        5              2085          4165            33.335 ns        13.901 us       27.768 us
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1  11              11             11            11              73.337 ns        73.337 ns       73.337 ns
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2         3 ~ 4100        3              1028          4100            20.001 ns        6.854 us        27.335 us
vadd_1        vadd         compute_conv                                                   19 ~ 4116       19             1044          4116            0.127 us         6.960 us        27.441 us
vadd_1        vadd         store_result_Pipeline_VITIS_LOOP_260_2                         undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         store_result                                                   undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         vadd                                                           undef           undef          undef         undef           undef            undef           undef

Area Information
Compute Unit  Kernel Name  Module Name                                                    FF    LUT   DSP  BRAM  URAM
------------  -----------  -------------------------------------------------------------  ----  ----  ---  ----  ----
vadd_1        vadd         entry_proc                                                     3     37    0    0     0
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2               42    89    0    0     0
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                           562   529   0    0     0
vadd_1        vadd         load_inputs                                                    871   1319  0    0     0
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0               339   294   0    0     0
vadd_1        vadd         create_conv_window                                             536   524   0    2     0
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1  303   142   0    0     0
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2         583   1375  0    0     0
vadd_1        vadd         compute_conv                                                   1023  1656  0    0     0
vadd_1        vadd         store_result_Pipeline_VITIS_LOOP_260_2                         495   522   0    0     0
vadd_1        vadd         store_result                                                   755   701   0    0     0
vadd_1        vadd         vadd                                                           6027  8117  0    2     0
--------------------------------------------------------------------------------------
```

Figure 16: Synthesis report for synthesising the minimum configuration.



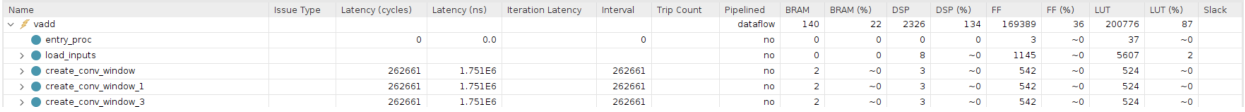
Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)
vadd							dataflow	2	~0	45	2	6027	1	8117	3
entry_proc		0	0.0		0		no	0	0	0	0	3	~0	37	~0
load_inputs							no	0	0	6	~0	871	~0	1319	~0
load_inputs_Pipeline_kernel_load_VITIS_LOOP_50_2		12	80.004		12		no	0	0	0	0	42	~0	89	~0
kernel_load_VITIS_LOOP_50_2		10	66.670	3	1	9	yes								
load_inputs_Pipeline_VITIS_LOOP_62_4							no	0	0	3	~0	562	~0	529	~0
image_load_VITIS_LOOP_62_4				74	1		yes								
create_conv_window		4165	2.777E4		4165		no	2	~0	3	~0	536	~0	524	~0
create_conv_window_Pipeline_update_conv_window_0		4163	2.776E4		4163		no	0	0	0	0	339	~0	294	~0
update_conv_window_0		4161	2.774E4	2	1	0~4160	yes								
compute_conv		4116	2.744E4		4116		no	0	0	30	1	1023	~0	1656	~0
compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1		11	73.337		11		no	0	0	0	0	303	~0	142	~0
load_coeffs_to_compute_VITIS_LOOP_208_1		9	60.003	2	1	9	yes								
compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2		4100	2.734E4		4100		no	0	0	27	1	583	~0	1375	~0
full_image_conv_VITIS_LOOP_219_2		4098	2.732E4	4	1	0~4096	yes								
store_result							no	0	0	6	~0	755	~0	701	~0
store_result_Pipeline_VITIS_LOOP_260_2							no	0	0	3	~0	495	~0	522	~0
output_store_VITIS_LOOP_260_2				72	1		yes								

Figure 17: Synthesis report estimates for synthesising the minimum configuration.



```
Latency Information
Compute Unit  Kernel Name  Module Name                                                    Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolute)  Worst (absolute)
------------  -----------  -------------------------------------------------------------  --------------  -------------  ------------  --------------  ---------------  --------------  ----------------
vadd_1        vadd         entry_proc                                                     0               0              0             0               0 ns             0 ns            0 ns
vadd_1        vadd         load_inputs_Pipeline_kernel_load_VITIS_LOOP_49_1_VITIS_LOOP_50_2  633          633            633           633             4.220 us         4.220 us        4.220 us
vadd_1        vadd         load_inputs_Pipeline_VITIS_LOOP_62_4                           undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         load_inputs                                                    undef           undef          undef         undef           undef            undef           undef
vadd_1        vadd         create_conv_window_Pipeline_update_conv_window_0               3 ~ 262659      3              131331        262659          20.001 ns        0.876 ms        1.751 ms
vadd_1        vadd         create_conv_window                                             5 ~ 262661      5              131333        262661          33.335 ns        0.876 ms        1.751 ms
vadd_1        vadd         compute_conv_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1  11              11             11            11              73.337 ns        73.337 ns       73.337 ns
vadd_1        vadd         compute_conv_Pipeline_full_image_conv_VITIS_LOOP_219_2         3 ~ 262148      3              65540         262148          20.001 ns        0.437 ms        1.748 ms
vadd_1        vadd         compute_conv                                                   19 ~ 262164     19             65556         262164          0.127 us         0.437 ms        1.748 ms
vadd_1        vadd         create_conv_window_1_Pipeline_update_conv_window_0             3 ~ 262659      3              131331        262659          20.001 ns        0.876 ms        1.751 ms
vadd_1        vadd         create_conv_window_1                                           5 ~ 262661      5              131333        262661          33.335 ns        0.876 ms        1.751 ms
vadd_1        vadd         compute_conv_2_Pipeline_load_coeffs_to_compute_VITIS_LOOP_208_1  11            11             11            11              73.337 ns        73.337 ns       73.337 ns
vadd_1        vadd         compute_conv_2_Pipeline_full_image_conv_VITIS_LOOP_219_2       3 ~ 262148      3              65540         262148          20.001 ns        0.437 ms        1.748 ms
vadd_1        vadd         compute_conv_2                                                 19 ~ 262164     19             65556         262164          0.127 us         0.437 ms        1.748 ms
vadd_1        vadd         create_conv_window_3_Pipeline_update_conv_window_0             3 ~ 262659      3              131331        262659          20.001 ns        0.876 ms        1.751 ms
```

Figure 18: Synthesis report for synthesising the maximum configuration (oversized)



Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)	FF	FF (%)	LUT	LUT (%)	Slack
vadd							dataflow	140	22	2326	134	169389	36	200776	87	
entry_proc		0	0.0		0		no	0	0	0	0	3	~0	37	~0	
load_inputs							no	0	0	8	~0	1145	~0	5607	2	
create_conv_window		262661	1.751E6		262661		no	2	~0	3	~0	542	~0	524	~0	
create_conv_window_1		262661	1.751E6		262661		no	2	~0	3	~0	542	~0	524	~0	
create_conv_window_3		262661	1.751E6		262661		no	2	~0	3	~0	542	~0	524	~0	

Figure 19: Synthesis report estimates for synthesising the maximum configuration (oversized)

Bibliography

[1] T. A. Johansen, 'Development of a small satellite with a hyperspectral imaging payload and onboard processing for ocean color', Available at https://folk.ntnu.no/torarnj/V1.2.0_Sivert_PhD_thesis.pdf, PhD Thesis, NTNU, 2022.

[2] J. Schott, *Remote Sensing: The Image Chain Approach* (Oxford Series on Optical and Imaging Sciences Series). Oxford University Press, 1997, ISBN: 9780195087260. [Online]. Available: https://books.google.no/books?id=DAh5bca5iYIC.

[3] Aappo Roos, *Hsi lwir stones*, This work is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/deed.en., 2011. [Online]. Available: https://commons.wikimedia.org/wiki/File:HSI_LWIR_stones.png.

[4] M. E. Grøtte, R. Birkeland, E. Honoré-Livermore *et al.*, 'Ocean color hyperspectral remote sensing with high resolution and low latency—the hypso-1 cubesat mission', *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–19, 2022. DOI: 10.1109/TGRS.2021.3080175.

[5] J. Zou, Y. Han and S.-S. So, 'Overview of artificial neural networks', in *Artificial Neural Networks: Methods and Applications*, D. J. Livingstone, Ed. Totowa, NJ: Humana Press, 2009, pp. 14–22, ISBN: 978-1-60327-101-1. DOI: 10.1007/978-1-60327-101-1_2. [Online]. Available: https://doi.org/10.1007/978-1-60327-101-1_2.

[6] I. El Naqa and M. J. Murphy, 'What is machine learning?', in *Machine Learning in Radiation Oncology: Theory and Applications*, I. El Naqa, R. Li and M. J. Murphy, Eds. Cham: Springer International Publishing, 2015, pp. 3–11, ISBN: 978-3-319-18305-3. DOI: 10.1007/978-3-319-18305-3_1. [Online]. Available: https://doi.org/10.1007/978-3-319-18305-3_1.

[7] S. Albawi, T. A. Mohammed and S. Al-Zawi, 'Understanding of a convolutional neural network', in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[8] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli and J. Sohl-Dickstein, 'On the expressive power of deep neural networks', in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 2847–2854. [Online]. Available: https://proceedings.mlr.press/v70/raghu17a.html.

[9] K. O'Shea and R. Nash, *An introduction to convolutional neural networks*, 2015. arXiv: 1511.08458 [cs.NE].

[10] A. Ajit, K. Acharya and A. Samanta, 'A review of convolutional neural networks', in *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, 2020, pp. 1–5. DOI: 10.1109/ic-ETITE47903.2020.049.

[11] H. Srivastava and K. Sarawadekar, 'A depthwise separable convolution architecture for cnn accelerator', in *2020 IEEE Applied Signal Processing Conference (ASPCON)*, 2020, pp. 1–5. DOI: 10.1109/ASPCON49795.2020.9276672.

[12] R. Nirthika, S. Manivannan, A. Ramanan and R. Wang, 'Pooling in convolutional neural networks for medical image analysis: A survey and an empirical study', *Neural Computing and Applications*, vol. 34, no. 7, pp. 5321–5347, Apr. 2022, ISSN: 1433-3058. DOI: 10.1007/s00521-022-06953-8. [Online]. Available: https://doi.org/10.1007/s00521-022-06953-8.

[13] A. Zhang, Z. C. Lipton, M. Li and A. J. Smola, 'Dive into deep learning', *arXiv preprint arXiv:2106.11342*, 2021.

[14] H. Ahn and C. Yim, 'Convolutional neural networks using skip connections with layer groups for super-resolution image reconstruction based on deep learning', *Applied Sciences*, vol. 10, no. 6, 2020, ISSN: 2076-3417. DOI: 10.3390/app10061959. [Online]. Available: https://www.mdpi.com/2076-3417/10/6/1959.

[15] Z. Wang, S. Liang, L. Xu, W. Song, D. Wang and D. Huang, 'Dimensionality reduction method for hyperspectral image analysis based on rough set theory', *European Journal of Remote Sensing*, vol. 53, no. 1, pp. 192–200, 2020. DOI: 10.1080/22797254.2020.1785949. eprint: https://doi.org/10.1080/22797254.2020.1785949. [Online]. Available: https://doi.org/10.1080/22797254.2020.1785949.

[16] H. Tulapurkar, B. Banerjee and B. K. Mohan, 'Effective and efficient dimensionality reduction of hyperspectral image using cnn and lstm network', in *2020 IEEE India Geoscience and Remote Sensing Symposium (InGARSS)*, 2020, pp. 213–216. DOI: 10.1109/InGARSS48198.2020.9358957.

[17] A. Uberoi, *Introduction to dimensionality reduction*, May 2023. [Online]. Available: https://www.geeksforgeeks.org/dimensionality-reduction/.

[18] D. Kapla, L. Fertl and E. Bura, 'Fusing sufficient dimension reduction with neural networks', *Computational Statistics & Data Analysis*, vol. 168, p. 107 390, 2022, ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2021.107390. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167947321002243.

[19] X. Zhang, Y. Guo and X. Zhang, 'Hyperspectral image classification based on optimized convolutional neural networks with 3d stacked blocks', *Earth Science Informatics*, vol. 15, no. 1, pp. 383–395, Mar. 2022, ISSN: 1865-0481. DOI: 10.1007/s12145-021-00731-1. [Online]. Available: https://doi.org/10.1007/s12145-021-00731-1.

[20] J. Cao and X. Li, *A 3d 2d convolutional neural network model for hyperspectral image classification*, 2021. arXiv: 2111.10293 [cs.CV]. [Online]. Available: https://doi.org/10.48550/arXiv.2111.10293.

[21] S. Che, J. Li, J. W. Sheaffer, K. Skadron and J. Lach, 'Accelerating compute-intensive applications with gpus and fpgas', in *2008 Symposium on Application Specific Processors*, 2008, pp. 101–107. DOI: 10.1109/SASP.2008.4570793.

[22] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno and P. H. Jones, 'Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels', in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8. DOI: 10.1109/ICESS.2019.8782524.

[23] A. Boutros, S. Yazdanshenas and V. Betz, 'You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference', *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018, ISSN: 1936-7406. DOI: 10.1145/3242898. [Online]. Available: https://doi.org/10.1145/3242898.

[24] X. Advanced Micro Devices. 'Vitis high-level synthesis user guide (ug1399)'. (), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls (visited on 14th Feb. 2023).

[25] Y. Herklotz, Z. Du, N. Ramanathan and J. Wickerson, 'An empirical study of the reliability of high-level synthesis tools', in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 219–223. DOI: 10.1109/FCCM51124.2021.00034.

[26] I. Skliarova, 'Analysis and comparison of different approaches to implementing a network-based parallel data processing algorithm', *Journal of Low Power Electronics and Applications*, vol. 12, no. 3, 2022, ISSN: 2079-9268. DOI: 10.3390/jlpea12030038. [Online]. Available: https://www.mdpi.com/2079-9268/12/3/38.

[27] X. Advanced Micro Devices. 'Vitis unified software platform documentation: Application acceleration development (ug1393)'. (), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration (visited on 16th Feb. 2023).

[28] S. Netteland, 'Exploration and implementation of large cnn models for image segmentation in hyperspectral cubesat missions', Available at https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3020478, Master's Thesis, NTNU, 2022.

[29] O. Ronneberger, P. Fischer and T. Brox, *U-net: Convolutional networks for biomedical image segmentation*, 2015. arXiv: 1505.04597 [cs.CV].

[30] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'Imagenet classification with deep convolutional neural networks', *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, ISSN: 0001-0782. DOI: 10.1145/3065386. [Online]. Available: https://doi.org/10.1145/3065386.

[31] X. Yu, Y. Wang, J. Miao *et al.*, 'A data-center fpga acceleration platform for convolutional neural networks', in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 151–158. DOI: 10.1109/FPL.2019.00032.