TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

FPGA-Based Embedded System to Detect Cracks in Harbor Structures with the Use of Convolutional Neural Network

---

Author:
Alexandros-Ioannis SIFAKIS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Sotirios IOANNIDIS
Asst. Prof. Grigorios TSAGKATAKIS(UoC)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

October 4, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

FPGA-Based Embedded System to Detect Cracks in Harbor Structures with the Use of Convolutional Neural Network

by Alexandros-Ioannis SIFAKIS

Convolutional Neural Networks are highly effective in a wide range of applications, particularly in the field of computer vision, where they perform very well at recognizing patterns and objects. One key application of CNNs is to detect cracks in harbor structures. In this thesis, an FPGA-based accelerator intellectual property core for this application was developed, based on a pre-existing Convolutional Neural Network. The accelerator will be integrated with a RISC-V computing core for real-time, on-site crack detection. The large memory footprint of the CNN weights (> 130MB) did not allow for all weights be be internally stored in the FPGA's 71MB space. To address this, extensive experimentation with the K-means algorithm was performed in order to effectively compress the floating point weights so that they would fit insid the FPGAs's memory. This was achieved, resulting in a 4X compression, while maintaining at least 95% accuracy vs. the reference CNN. The CNN, a UNET of no less than 24 stages, was designed, synthesized and functionally verified with the Xilinx Vitis HLS CAD tool suite. Functional verification was completed successfully, using the same dataset that was used to train and evaluate the model. The IP core can be integrated as-is with the RISC V processor on the Alveo U55C system, however, several changes have been proposed in this thesis to improve performance.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

FPGA-Based Embedded System to Detect Cracks in Harbor Structures with the Use of Convolutional Neural Network

by Alexandros-Ioannis SIFAKIS

Τα συνελικτικά νευρωνικά δίκτυα είναι ιδιαίτερα αποτελεσματικά σε ένα ευρύ φάσμα εφαρμογών, ιδίως στον τομέα της όρασης υπολογιστών, όπου αποδίδουν πολύ καλά στην αναγνώριση μοτίβων και αντικειμένων. Μια βασική εφαρμογή των CNN είναι η ανίχνευση ρωγμών σε λιμενικές κατασκευές. Στην παρούσα διατριβή, αναπτύχθηκε ένας πυρήνας πνευματικής ιδιοκτησίας επιταχυντή βασισμένος σε FPGA για την εφαρμογή αυτή, βασισμένος σε ένα προϋπάρχον Συνελικτικό Νευρωνικό Δίκτυο. Ο επιταχυντής θα ενσωματωθεί με έναν υπολογιστικό πυρήνα RISC-V για την επιτόπια ανίχνευση ρωγμών σε πραγματικό χρόνο. Το μεγάλο αποτύπωμα μνήμης των βαρών του CNN ($>$ 130MB) δεν επέτρεψε την εσωτερική αποθήκευση όλων των βαρών στον χώρο των 71MB της FPGA . Για να αντιμετωπιστεί αυτό, πραγματοποιήθηκε εκτεταμένος πειραματισμός με τον αλγόριθμο K-Means προκειμένου να συμπιεστούν αποτελεσματικά τα βάρη κινητής υποδιαστολής ώστε να χωρέσουν στη μνήμη της FPGA . Αυτό επιτεύχθηκε, με αποτέλεσμα τη συμπίεση κατά 4X, ενώ διατηρήθηκε τουλάχιστον 95% ακρίβεια σε σχέση με το CNN αναφοράς. Το CNN , ένα UNet όχι λιγότερο από 24 στάδια, σχεδιάστηκε, συντέθηκε και επαληθεύτηκε λειτουργικά με τη σουίτα εργαλείων CAD Vitis HLS της Xilinx . Η λειτουργική επαλήθευση ολοκληρώθηκε με επιτυχία, χρησιμοποιώντας το ίδιο σύνολο δεδομένων που χρησιμοποιήθηκε για την εκπαίδευση και την αξιολόγηση του μοντέλου. Ο πυρήνας IP μπορεί να ενσωματωθεί ως έχει με τον επεξεργαστή RISC-V στο σύστημα Alveo U55C , ωστόσο, στην παρούσα διατριβή προτείνονται διάφορες αλλαγές για τη βελτίωση της απόδοσης.

Acknowledgements

I would like to express my deepest gratitude to my professor and primary advisor, Apostolos Dollas, for his unwavering belief in my ability to complete this thesis, for giving me the opportunity to undertake it, and for his invaluable guidance and support throughout its development. I am also thankful to him for introducing me to the Foundation for Research and Technology (FORTH). My sincere thanks go to Assistant Professor Grigorios Tsagkatakis for his insightful suggestions and guidance throughout my work. I am equally grateful to Professor Sotirios Ioannidis for serving as the third member of my committee and for dedicating his time to review this thesis.

I would also like to acknowledge the FORTH Institute for welcoming me and providing the ideal environment for my research. A special note of appreciation goes to Manolis Perakis for his exceptional assistance with tools, contributions, and design insights, and to Pantelis Xirouchakis, whose vast knowledge and experience in hardware helped me overcome numerous challenges. Their support was crucial in bringing this thesis to completion.

Finally, I am deeply thankful to my friends and family, whose constant encouragement and belief in me provided the strength and motivation I needed to see this work through to the end. Their unwavering support has been invaluable.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CLB	**C**onfigurable **L**ogic **B**lock
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessor **U**nit
DSP	**D**igital **S**ignal **P**rocessor
DHM	**D**igital **H**olography **M**emory
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
IDE	**I**ntegrated **D**evelopment **E**nvironment
LUT	**L**ook **U**p **T**able
ILVRC	**I**mageNet **L**arge-Scale **V**isual **R**ecognition **C**hallenge
IoU	**I**ntersection **o**ver **U**nion
PTQ	**P**ost-**T**raining **Q**uantization
QAT	**Q**uantization **A**ware **T**raining
REsNet	**R**esidual **N**etwork
RTL	**R**egister **T**ransfer **L**evel
SIMO	**S**ingle **I**nput **M**ultiple **O**utput
SNR	**S**ignal-to-**N**oise **R**atio
SM	**S**treaming **M**ultiprocessor
SoCs	**S**ystem **o**n **C**hips
SP	**S**treaming **P**rocessor
TPU	**T**ensor **P**rocessing **U**nit
USV	**U**nmanned **S**urface **V**ehicle
VGG	**V**isual **G**eometry **G**roup

Dedicated to my family and friends…

Chapter 1

Introduction

Convolutional Neural Networks(CNN) are a key tool in AI, especially in areas like computer vision and natural language processing. Their ability to detect features from raw data makes them essential for tasks like facial recognition, autonomous driving, and medical imaging. However, as CNNs become more complex, they require significant computational power to handle intensive matrix operations and real-time data processing.

To meet the increasing demand for speed and efficiency, hardware accelerators specifically designed for CNNs have emerged. These accelerators, including Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs), are optimized to handle the massive parallelism and computational load of CNNs far more efficiently than traditional CPUs.

GPUs are widely used for CNN acceleration due to their parallel processing architecture, designed to handle large-scale matrix operations like convolutional layers. With thousands of efficient cores, GPUs are ideal for training deep neural networks, especially with large datasets and complex data.

FPGAs, on the other hand, offer flexibility and power efficiency. Unlike GPUs, which are fixed in their architecture, FPGAs are reconfigurable, meaning they can be programmed to implement specific tasks for CNN inference with great efficiency. This reconfigurability allows for customized data flow, making FPGAs highly adaptable to specific CNN architectures. Furthermore, they consume less power than GPUs, making them suitable for edge devices and embedded systems where energy consumption is a concern.

ASICs are custom-designed chips optimized for specific tasks, making them the most power-efficient and fastest option for CNN acceleration. Unlike

GPUs and FPGAs, which offer flexibility, ASICs are fixed and cannot be reprogrammed. However, their specialization makes them ideal for large-scale deployments in cloud data centers. A key example is Google's Tensor Processing Unit (TPU), an ASIC built to accelerate machine learning, especially CNNs.

In essence, CNNs are revolutionizing a wide range of industries, and hardware accelerators are enabling them to perform at the speed necessary to handle the demands of real-time, large-scale applications. This synergy between CNNs and hardware accelerators is driving innovation across AI and computational fields.

1.1 Motivation

As previously discussed, both TPUs and GPUs represent excellent options for accelerating CNNs. However, in this particular thesis, FPGAs present certain advantages that make them a more suitable choice. In the context of an unmanned surface vehicle, the system must not only be capable of rapid processing but also exhibit low power consumption. The use of a TPU on board would be an inefficient use of energy, while reliance on communication with server-based TPUs would result in unessecary latency.

FPGAs offer a significant speedup while consuming less power. They can be deployed locally on the Unmanned Surface Vehicle(USV), eliminating the need for external servers. The proposed system integrates an FPGA guided by a RISC-V CPU, with external memory serving as the connection point between them.To optimize performance, the external memory will only be used for image exchange, not for loading CNN weights or other essential data during processing. Weight quantization may be necessary to ensure all computations take place in internal memory, avoiding potential bottlenecks caused by memory exchanges.

The objective of this thesis is to design and build an IP core for the CNN provided by Almende, for integration with a RISC-V processor in an integrated system. The IP core will ensure that the CNN operates internally on the FPGA, with external communication limited to image input and output, while prioritizing energy efficiency.

1.2 Scientific Contributions

This thesis focuses on analyzing a CNN and transforming it into an accelerated version capable of running independently on the FPGA, specifically the Alveo U55C. The objective is to optimize the CNN for improved performance on the FPGA, eliminating the need for constant communication with external systems. By running the accelerator directly on the FPGA, data transfers between the CNN and other components are minimized, which leads to significant improvements in both energy efficiency and latency. The elimination of extra reads and writes reduces bottlenecks, ensuring faster computation times and lower power consumption, making this approach ideal for applications where speed and efficiency are critical.

The contributions on this thesis can be divided on the following:

- A comprehensive study of the CNN developed by Almende, specifically a 40-stage UNET, was conducted to explore its suitability for FPGA-based implementation.
- The CNN was modeled for FPGA deployment by converting the original TensorFlow Keras functions into hardware description language (HDL) using specialized tools. Based on this model, an FPGA accelerator was designed using AMD Xilinx's Vitis HLS tool. Functional verification, testing, and benchmarking were conducted using Python and C++ libraries, ensuring accuracy and consistency by using the same datasets as the original UNET throughout the process.
- Key challenges and limitations of implementing the CNN on an FPGA were identified, particularly the high memory demand of 130 MB (Mega Bytes) for weights, exceeding the target FPGA's 71 MB of internal BRAM. This memory constraint significantly limits parallel processing capabilities.
- Weight quantization was implemented to reduce the memory footprint, nearly halving the model size to fit within FPGA constraints. This optimization maintained high model performance, achieving 99% accuracy in crack detection and 90% accuracy in crack segmentation.
- The final accelerator design was optimized for size, ensuring compatibility for integration with a RISC-V processor, meeting the requirements for deployment in an embedded system.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** This chapter provides an introduction to Machine Learning, with a focus on Convolutional Neural Networks. It explores their structure, the layers they utilize, and the various types of CNN architectures.
- **Chapter 3 - Related Work:** This chapter reviews relevant research in areas related to this thesis, including FPGA and GPU acceleration, CNN quantization, crack segmentation applications, and the use of U-Net architectures.
- **Chapter 4 - Modeling:** In this chapter, the detailed structure of the provided CNN is presented, along with algorithms for its functions. The rationale behind selecting the K-means parameters is also explained.
- **Chapter 5 - FPGA Implementation:** This chapter outlines the system architecture where the accelerator is applied. It also details the tools used, along with descriptions of the algorithm implementations in C.
- **Chapter 6 - Results:** The chapter presents the performance results of the hardware accelerator, including both latency measurements and the generated images. A discussion of these results is also included.
- **Chapter 7 - Conclusions and Future Work:** The final chapter summarizes the contributions of this thesis, evaluates its success, and suggests potential areas for further improvement and future work.

Chapter 2

Theoretical Background

In this chapter a few concepts about the CNNs and AI in general are described so every reader can have the theoretical background to understand this thesis.

2.1 Machine Learning

The basic concept of machine learning in data science involves using statistical learning and optimization methods that let computers analyze datasets and identify patterns. Machine learning techniques leverage data mining to identify historic trends and inform future models. In supervised machine learning, algorithms typically comprise three main components. The first is a decision process, which involves a series of calculations or steps that analyze the data and make an initial prediction about the pattern being sought. Next is an error function, which measures the accuracy of this prediction by comparing it to known examples when available. This function assesses how closely the predicted outcome matches the actual result, providing a way to quantify any discrepancies. Finally, there is an updating or optimization process, where the algorithm refines its decision-making approach based on the errors identified. This iterative process aims to minimize future errors, improving the model's accuracy with each cycle. [1]

2.1.1 Neural Network

A neural network is a machine learning model that emulates the decision-making processes of the human brain by replicating the way biological neurons interact to recognize patterns, evaluate alternatives, and reach conclusions. A neural network is composed of layers of interconnected nodes, or artificial neurons, which include an input layer, one or more hidden layers, and an output layer. Each node is connected to other nodes and is assigned a specific weight and threshold value. When the output of a node exceeds its threshold, it becomes activated, transmitting data to the next layer in the network. If the output does not meet the threshold, the data is not forwarded, effectively halting the flow of information to subsequent layers. [2]

2.1.2 Convolution Neural Networks

A CNN is a type of neural network used primarily for image recognition and processing, due to its ability to recognize patterns in images. A CNN is a powerful tool but requires millions of labelled data points for training. It is a a specific implementation of machine learning models designed for tasks where spatial structure is important (like images).[3]

Layer Architecture

There can be a lot of available architectures to form a CNN, a simple and common architecture is:

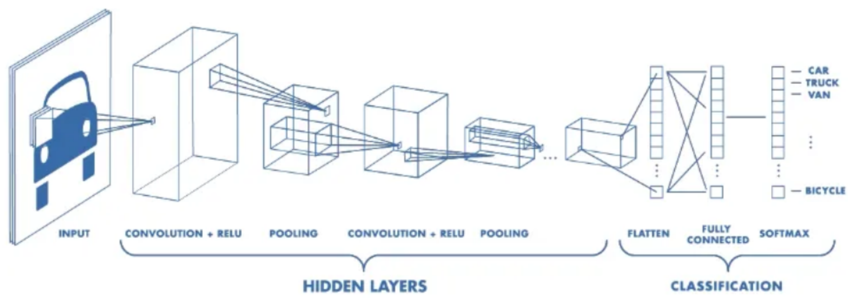


FIGURE 2.1: Convolutional neural network model
https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

As it can be seen in the image there are a lot of different types of layers.

Convolution Layer

The convolution layer serves as the fundamental building block of a Convolutional Neural Network (CNN) and is typically the first layer in its architecture. Several variations of this layer exist [4], each with unique characteristics and applications:

- **2D Convolution**

  The 2D convolution layer, commonly abbreviated as Conv2D, is the most frequently used type of convolution. It applies a filter or kernel that "slides" over the 2D input data, performing element-wise multiplication followed by a summation to produce a single output pixel. This process is repeated for each location over which the kernel slides, transforming one 2D feature matrix into another.

- **Dilated or Atrous Convolution**

  Dilated, or atrous convolution, enlarges the receptive field of the kernel without increasing the number of parameters by inserting zero values (dilations) between the kernel weights. This approach is particularly effective in real-time applications or scenarios with limited computational resources, as it reduces the demand on memory and processing power.

- **Separable Convolution**

  Separable convolutions come in two main types: spatial separable convolutions and depthwise separable convolutions. Spatial separable convolutions focus on the spatial dimensions (width and height) of the input and kernel, while depthwise separable convolutions apply more complex operations where kernels cannot be "factored" into smaller components. Depthwise separable convolutions are more commonly used due to their efficiency and reduced computational cost.

- **Transposed Convolution**

  Also known as deconvolution or fractionally strided convolution, transposed convolution layers perform a regular convolution operation in reverse, effectively upscaling the input feature map by reversing the spatial transformation applied by standard convolutions.

Activation Function

In the context of neural networks, an activation function is a mathematical function applied to the output of a neuron, crucial for enabling the network to learn and represent complex patterns in data. By introducing non-linearity into the model, activation functions prevent the network from merely behaving like a linear regression model, regardless of its depth or the number of layers. The activation function determines whether a neuron should be activated by computing the weighted sum of its inputs and adding a bias. This process allows the network to make decisions about which neurons to activate, contributing to the network's ability to model intricate relationships within the data. Moreover, activation functions are essential for gradient-based training, as they provide the gradients necessary for the backpropagation algorithm, which adjusts the weights and biases during the learning process. There are a lot of activations functions few of them are:[5]

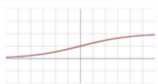
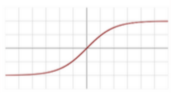
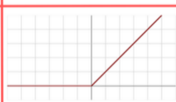
ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		f(x) = x	f'(x) = 1	$(-\infty, \infty)$
Binary Step		f(x) = $\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	f'(x) = $\begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		f(x) = $\sigma(x) = \frac{1}{1+e^{-x}}$	f'(x) = $f(x)(1-f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		f(x) = $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	f'(x) = $1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		f(x) = $\begin{cases} 0 & \text{if } x<0 \\ x & \text{if } x \geq 0 \end{cases}$	f'(x) = $\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$

FIGURE 2.2: Different activation functions
https://iq.opengenus.org/linear-activation-function/

- **Linear**

  The linear activation function in neural networks is a simple function that outputs the input value itself. It does not introduce any non-linearity into the network, which can limit its ability to learn complex patterns. While it can be useful in certain situations, such as regression tasks, it is generally not used in deep neural networks due to the vanishing gradient problem. Other activation functions, such as the sigmoid, tanh, and ReLU functions, are more commonly used in deep learning.

- **Binary Step**

  The binary step activation function is a simple function that returns 1 if the input is positive and 0 if the input is negative. It's like a switch that turns on or off based on the input. It has some limitations, such as being non-differentiable and having limited expressiveness. However, it can be useful in certain scenarios where a simple on/off decision is needed.

- **Sigmoid**

  The sigmoid activation function is a mathematical function that maps any real-valued input to a value between 0 and 1. The sigmoid function has a characteristic S-shape, which makes it useful for representing probabilities. When the input is very negative, the output is close to 0. As the input increases, the output approaches 1. This makes it a good choice for modeling probabilities because it ensures that the output is always between 0 and 1, which is the range of probabilities. However, the sigmoid function has some drawbacks. One is that it can suffer from the vanishing gradient problem, which can make it difficult to train deep neural networks. Another is that it is not centered around 0, which can make training more challenging.

- **Hyperbolic Tangent**

  The tanh (hyperbolic tangent) activation function is a mathematical function that maps any real-valued input to a value between -1 and 1. It's a common choice for the hidden layers of neural networks, as it can help to normalize the outputs of neurons, which can improve the training process.The tanh function has a similar shape to the sigmoid function, but it is centered around 0, which can make it a better choice for certain applications. Additionally, the tanh function can help to prevent the vanishing gradient problem, which can occur in deep neural networks. However, the tanh function also has some drawbacks, such as being computationally more expensive than the ReLU activation function.

- **Rectified Linear Unit**

  ReLU is the most popular activation function in deep learning. It's a simple function that outputs the input directly if it's positive, and 0 if it's negative. This helps to prevent the vanishing gradient problem and makes training deep neural networks more efficient. However, ReLU can suffer from the "dying ReLU" problem, where neurons can become permanently inactive.

Pooling methods

Following the convolution layer and the activation function comes the third stage, we use a pooling function to modify the output of the layer further. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.In all cases, pooling helps to make the representation approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. [6]

Two widely used methods for pooling in neural networks are average pooling and max pooling. Max pooling is an operation that selects the maximum value from each region of the feature map covered by the filter. Consequently, the output of a max-pooling layer is a feature map that highlights the most prominent features from the previous feature map.

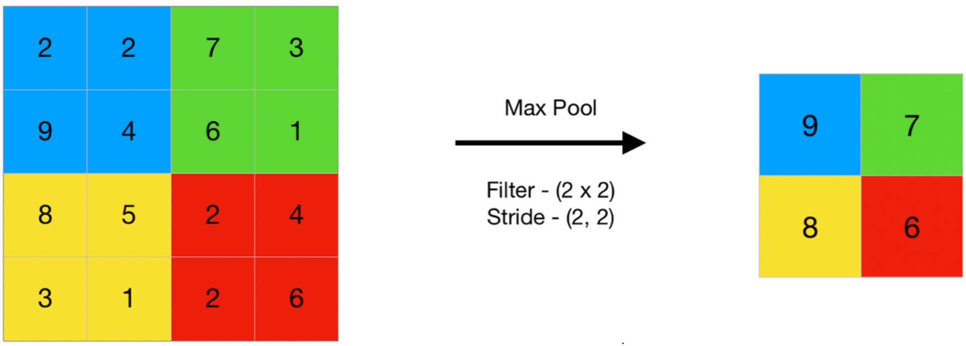


FIGURE 2.3: Max pooling method
https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

In contrast, average pooling computes the average value of the elements within each region of the feature map covered by the filter. While max pooling focuses on retaining the most significant feature within a given region, average pooling provides a more generalized representation by averaging all features present in the region.

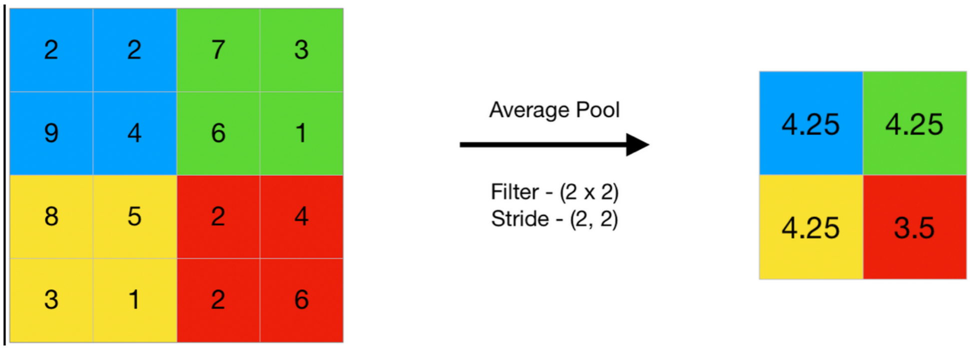


FIGURE 2.4: Average pooling method
https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

Fully connected layers

The fully connected layer, or dense layer, is a crucial component in CNNs, responsible for capturing global patterns and relationships in the input data. Positioned at the end of the network architecture, after the convolutional and pooling layers, it connects every neuron from the previous layer to every neuron within itself. This extensive connectivity enables the layer to learn complex, non-linear mappings between inputs and outputs, facilitating high-level reasoning and decision-making. By receiving inputs from all preceding neurons, each neuron in the fully connected layer applies a set of weights and biases, followed by an activation function, to produce its output. This allows the network to identify intricate relationships within the data, making predictions based on a combination of multiple features rather than relying on individual ones. For example, in a CNN trained to classify handwritten digits, the fully connected layer combines abstracted features like edges and textures to determine the specific digit in the image. [7]

Types of CNNs

By combining various architectural styles, several specialized CNNs have been developed, each distinguished by unique characteristics that cater to specific tasks or improve performance in certain aspects. These specialized CNNs have been tailored to address particular challenges in image processing, pattern recognition, and other domains. Some notable types of these specialized CNNs include:

- Unet

  U-Net is a CNN architecture specifically designed for image segmentation tasks, particularly in biomedical image analysis. Developed in 2015 at the University of Freiburg, U-Net has become one of the most widely used architectures for pixel-wise image segmentation due to its simplicity, efficiency, and effectiveness in producing high-quality segmentation results.[8]

- Resnet

  A Residual Neural Network (ResNet) is a deep learning architecture that introduces the concept of residual learning. Instead of learning the desired underlying mapping directly, ResNet layers learn residual functions with reference to the layer inputs. This is achieved through skip connections or shortcut connections that bypass one or more layers. This approach helps mitigate the vanishing gradient problem, allowing for the training of very deep networks. [9]

- VGGNet

  VGGNet is a type of convolutional neural network (CNN) architecture that was proposed by the Visual Geometry Group (VGG) from the University of Oxford in 2014. The model gained significant attention and recognition after it achieved top performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014. VGGNet is known for its simple and effective design, which relies on using very deep networks with small convolutional filters, making it one of the most influential and widely used CNN architectures in the field of computer vision. [10]

Quantization

Sometimes, the weights of neural networks, such as those in CNNs, are too large to meet performance requirements, posing a challenge for deployment. However, there are ways to reduce the storage size of these weights without significantly impacting performance. One effective method is quantization, which reduces the precision of numerical representations for model parameters and activations by lowering the number of bits used. This approach is especially valuable for deploying neural networks on resource-constrained devices, such as microcontrollers, as it minimizes memory usage and computational demands, enabling efficient inference.

Fro example, the process of quantization is the process of taking a neural network, which generally uses 32-bit floats to represent parameters, and instead converts it to use a smaller representation, like 8-bit integers. Going from 32-bit to 8-bit, for example, would reduce the model size by a factor of 4, so one obvious benefit of quantization is a significant reduction in memory. Or it can group some numbers and represent them with a smaller in capacity number.

There are several methods for quantizing a neural network, with the two most popular being Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). [11]

Quantization-Aware Training prepares a model for quantization during training by simulating reduced precision effects in both forward and backward passes using fake-quantization modules. This helps the model learn to be robust against quantization errors, bridging the gap between high-precision training and low-precision deployment. However, QAT increases training complexity due to the need for additional operations, adjustments to the loss function, and modifications to model layers. These changes make the training process more computationally intensive and require careful implementation, validation, and fine-tuning to minimize accuracy loss.

In contrast, Post-Training Quantization applies reduced precision only after the model is fully trained. PTQ does not account for quantization effects during training, focusing instead on compressing the pre-trained model for deployment on resource-constrained devices. Although less complex than QAT, PTQ may still result in accuracy loss since it directly compresses the model without prior adaptation. Thus, achieving the optimal balance between size reduction and accuracy may require careful calibration, especially in applications where high precision is essential.

Quantization can also be classified into two types based on how the quantization levels are defined: uniform and non-uniform. [12]

Uniform quantization divides the input range into equally spaced levels, making it simple to implement and efficient for hardware like CPUs and GPUs. However, it can cause significant quantization error when the data distribution is uneven, as it applies the same precision across the entire range.

Non-uniform quantization uses levels with unequal step sizes, better suited for data with non-uniform distributions, such as values clustered around zero. By allocating more levels to dense regions, it reduces quantization error and often maintains higher accuracy. However, it requires more complex algorithms and increases computational overhead.

Non-uniform PTQ is often the most useful approach because most neural networks are already pre-trained, and their weights typically follow a non-uniform distribution. Below are some examples of non-uniform PTQ methods that can be applied to the weights of a neural network.

- K-means K-means quantization is a method derived from the k-means clustering algorithm. It partitions a set of observations into (k) clusters, where each observation belongs to the cluster with the nearest mean (centroid). This technique is often used in image compression and color quantization, where it reduces the number of colors in an image while maintaining visual similarity. As the method used in this thesis, it will be analyzed to a later chapter. [13]
- Logarithmic Quantization Logarithmic quantization maps values to a logarithmic scale, which is particularly useful for compressing data with a wide dynamic range. This method is often used in neural network training to reduce the computational footprint while maintaining accuracy. It is effective because it reduces the bit-width of the data, making it more hardware-friendly. [14]
- Vector Quantization Vector quantization is a classical quantization technique used for data compression. It works by dividing a large set of points (vectors) into groups, each represented by a centroid. Vector quantization is a classical quantization technique used for data compression. It works by dividing a large set of points (vectors) into groups, each represented by a centroid. [15]
- Companding Companding is a technique that combines compression and expanding to achieve non-uniform quantization. It improves the signal-to-noise ratio (SNR) of weak signals by compressing the signal before quantization and expanding it after quantization. This method is commonly used in audio signal processing to reduce the effects of noise. [16]

Chapter 3

Related Work

The following chapter presents a series of case studies that examine a range of aspects related to the central theme of this thesis. Each case study presents a different method or technology used in the field, with a particular focus on accelerating convolutional neural networks designed for crack segmentation. The initial set of papers investigates the acceleration of CNNs on FPGAs and GPUs, offering a comparative analysis of their performance. Subsequently, a paper is presented that discusses the application of neural networks for crack segmentation, specifically on FPGAs. Finally, the last two papers introduce innovative approaches, including a pioneering implementation of a UNet on an FPGA and a particular quantization technique for a CNN.

3.1 CNN acceleration

3.1.1 Types of acceleration

Three types of acceleration are mentioned, as they are the most commonly used. The CPU method is available, but its results cannot be directly compared to the other three, so it is no included.

First are FPGAs, which are programmable hardware devices particularly well-suited for accelerating CNNs due to their ability to provide parallelism and energy efficiency. The architecture of an FPGA consists of configurable logic blocks (CLBs), a programmable interconnection network, digital signal processing (DSP) blocks, block RAM (BRAM), and other components that can be customized to execute specific CNN operations more efficiently than general-purpose processors, such as CPUs or GPUs. By leveraging these components, FPGA-based accelerators can achieve a significant performance boost for the key operations involved in CNNs. [17]

Secondly are GPUs, which can accelerate Convolutional Neural Networks (CNNs) by leveraging their parallel architecture to perform the many small computations required for convolutions efficiently. Each GPU contains multiple Streaming Multiprocessors (SMs), each with several Streaming Processors (SPs) that can perform concurrent computations, reducing the time needed for the intensive matrix multiplications required by CNNs. Additionally, CUDA programming optimizes these operations by converting convolutions into matrix multiplications, which are well-suited for GPUs. The use of shared memory also minimizes data transfer latency, further enhancing performance. This ability to handle large-scale parallel computations makes GPUs ideal for speeding up CNN workloads. [18]

Lastly, TPUs excel at accelerating CNNs due to their parallel architecture and specialized memory. Their 2D array of processing elements efficiently performs matrix multiplications and convolution operations. On-chip caching and dedicated memory reduce data transfers, improving throughput. TPUs also offer high bandwidth and compute capacity, making them well-suited for large CNN workloads. [19]

FPGA acceleration

To accelerate a CNN using FPGA-based accelerators, several key strategies exist and several papers that that explore them.

Key CNN operations can be optimized for FPGA acceleration. Convolution (CONV) operations, being the most computationally intensive and often accounting for over 90% of total computations, have been a focus of several studies. Efficient FPGA accelerators maximize parallelism by unrolling loops and utilizing multiple processing elements (PEs) to perform concurrent Multiply-Accumulate (MAC) operations. Activation functions like ReLU, often favored for their simplicity, are efficiently implemented on FPGAs, reducing complexity compared to alternatives like sigmoid or tanh. Additionally, pooling operations, which reduce feature map dimensions, can be performed efficiently with simple arithmetic units. Fully connected layers, known for their high memory demands, are optimized by minimizing memory accesses through techniques such as on-chip buffering and pipelining.

Numerous papers also highlight the use of compression techniques to reduce the computational and memory footprint of CNNs on FPGAs. Pruning and quantization are widely applied to decrease the number of connections and use lower-bit representations for weights and activations, respectively, reducing memory usage and speeding up calculations. Techniques like singular value decomposition (SVD) are also applied to reduce the dimensionality of weight matrices, and other approaches like weight sharing and Huffman coding further compress weights to make more efficient use of the FPGA's limited on-chip memory.

Efficient memory utilization is critical for FPGA performance, and several works address this through on-chip memory optimization and data reuse. By leveraging BRAMs for caching frequently accessed data, these approaches minimize costly off-chip memory access. Additionally, optimizing data reuse via loop rearrangement and techniques like loop tiling reduces redundant memory accesses, which improves both performance and power efficiency. [20]

FlexFlow, as discussed in various studies, adapts to different parallelism types in CNN layers, optimizing data paths and processing styles to meet each layer's needs. This flexibility is key to handling diverse CNN workloads.

FPGA dynamic reconfiguration, highlighted in several papers, allows layers to be reprogrammed on-the-fly, reducing downtime and maximizing resource use—offering an edge over static accelerators. High-level synthesis (HLS) tools also simplify FPGA accelerator design in languages like C or C++, cutting development time and enabling advanced parallelism and memory optimizations for improved performance.

Specialized FPGA architectures like neuFlow and MAPLE are frequently cited for their contributions to deep learning acceleration. These architectures employ highly parallel processing elements, customized memory hierarchies, and efficient dataflow mechanisms to accelerate CNN operations. By optimizing both computation and data movement, they achieve higher performance per watt compared to GPU-based accelerators. Techniques such as reduced bit-width for operations, on-chip memory utilization, and minimizing data transfers are critical to this efficiency.

Overall, research into FPGA-based CNN acceleration highlights the importance of leveraging parallel computation, dynamic reconfiguration, and memory optimization. Through these strategies, FPGA accelerators deliver significant speedups and energy savings, making them ideal for deploying deep learning models in real-time, resource-constrained environments. [21] [22]

GPU acceleration

The implementation of convolutional neural networks (CNNs) on GPUs offers significant performance improvements by leveraging the parallel processing capabilities of NVIDIA's CUDA framework. CNNs are computationally intensive, particularly during training, and the use of GPUs allows for substantial reductions in processing time, with speedups ranging from two to twenty-four times compared to CPU-based implementations.

GPU acceleration is especially beneficial for tasks such as convolutions and matrix multiplications, which can be efficiently parallelized. CUDA enables these operations by distributing them across multiple GPU cores, while libraries like CUBLAS enhance the efficiency of matrix computations. Memory optimization techniques, such as coalesced memory access and minimizing CPU-GPU data transfer, further contribute to improved performance. [18]

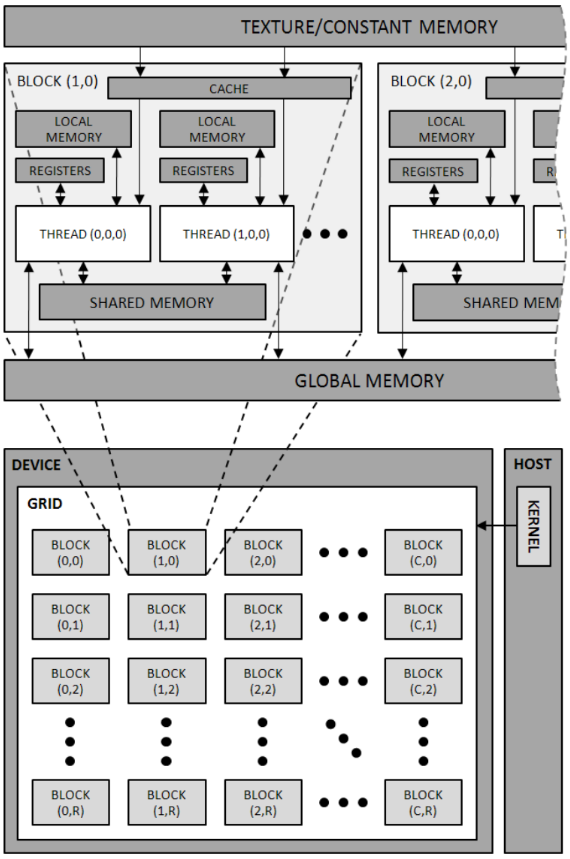


FIGURE 3.1: CUDA threading and memory topology.
https://web.archive.org/web/20170809034854id_/http://www.dps.uibk.ac.at/~klaus/Klaus_Kofler_-_Institute_for_Computer_Science_files/GPUCNN.pdf

A key optimization is the unfolding technique, which linearizes convolutions and allows them to be processed as matrix multiplications, improving data access patterns and GPU execution. Backpropagation is also optimized by pushing errors backward through the network, streamlining the training process.

Benchmarks on networks such as LeNet-5 and SimardNet show significant performance gains, with the GPU implementation consistently outperforming optimized CPU versions. The GPU approach also scales more effectively as input size and network complexity increase, making it well-suited for larger, more complex datasets.

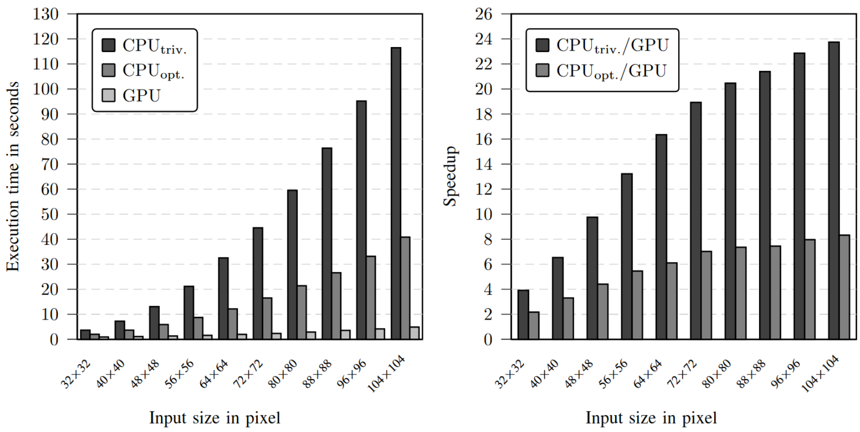


FIGURE 3.2: Execution time left and corresponding speedup right of the three different implementations(CPU optimized, CPU trivial and GPU) performing 1,000 learning iterations of a LeNet5.

https://web.archive.org/web/20170809034854id_/http://www.dps.uibk.ac.at/~klaus/Klaus_Kofler_-_Institute_for_Computer_Science_files/GPUCNN.pdf

Challenges remain, such as the higher power consumption of GPUs, but the substantial speedup achieved often results in lower overall energy use. The potential of GPU-accelerated CNNs is clear, offering faster and more scalable solutions for computationally intensive tasks like image recognition. [23] [24]

3.1.2 Comparison FPGA versus GPU

From the above it can be seen that both FPGAs and GPUs meet the requirements for accelerating CNNS. Therefore, the following paper will provide a comparison between the two so that the pros and cons, if not a winner between the two, can be made clear.

It is well known that GPUs are the dominant architecture for deep learning tasks due to their ability to handle large-scale parallelism with Single Instruction, Multiple Data (SIMD) execution. They excel at performing highly parallel tasks such as training and inference in CNNs, using high-level programming tools such as CUDA and libraries such as PyTorch for optimisations.

FPGAs, on the other hand, are gaining momentum in deep learning acceleration, particularly for tasks such as inference. FPGAs are known for their flexibility and power efficiency. Direct Hardware Mapping (DHM) techniques are used to map CNNs onto FPGA hardware, allowing the FPGA's resources to be used by storing weights and intermediate data on-chip, thus reducing memory access overhead.

FPGA with DHM involves fixed-point computation, which reduces memory complexity for features and weights. This allows FPGAs to reduce power consumption by using dedicated logic for each operation. The DHM approach uses a pipelined architecture where intermediate data and weights are stored locally within the FPGA to minimise external memory access, improve power efficiency and reduce execution time.

GPUs, on the other hand, rely on a hierarchical memory model that introduces latency based on memory accesses. While GPUs perform well for batch processing, they are less efficient when the tiers involve memory-intensive operations such as retrieving weights from external memory.

The paper concludes that while FPGAs are more energy-efficient and sometimes faster for smaller CNN layers, they are constrained by resource limitations. Therefore, heterogeneous systems that combine FPGA and GPU resources provide the best of both worlds: the energy efficiency of FPGA and the processing power of GPU. This is particularly beneficial for embedded deep learning systems like mobile or IoT devices.

Although FPGAs are highly efficient in terms of energy and latency for small-scale CNN tasks, the resource-intensive nature of DHM imposes constraints. FPGAs can only handle a limited number of layers and parameters, which limits their ability to handle state-of-the-art CNNs in their entirety. As CNNs grow deeper, the limited logic resources in FPGAs become a bottleneck, making it difficult to fully implement larger networks. [25] [26]

There is also a paper that talks about combining both FPGAs and GPUs to get the best acceleration on the CNNs, because only the advantages of both classes are taken into account. [27]

3.2 Crack Segmentation in FPGA

The next paper proposes a method for performing real-time semantic segmentation for autonomous vehicles using FPGAs. It leverages a compressed version of the ENet convolutional neural network architecture.

The authors utilize hls4ml, an open-source library that converts high-level machine learning models into hardware description code (HLS), allowing them to run on FPGAs. Originally developed for physics research, hls4ml was adapted here for semantic segmentation tasks in autonomous driving. The framework automates the creation of hardware designs from neural networks, significantly simplifying the process. Instead of manually coding hardware logic, designers work at a high level, and hls4ml handles the conversion to FPGA-compatible code. This allows for rapid prototyping and testing.

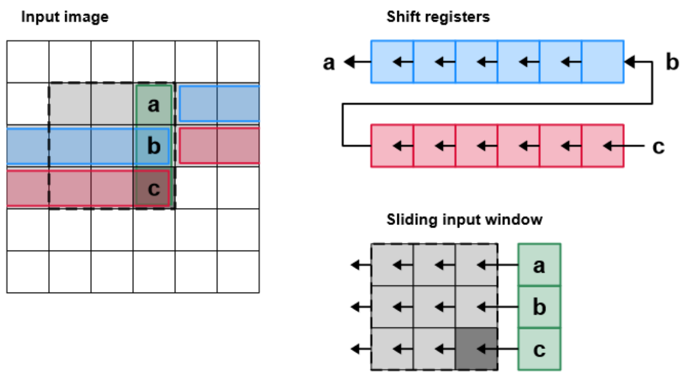


FIGURE 3.3: Schematic representation of the new hls4ml implementation of Convolutional layers
https://arxiv.org/pdf/2205.07690

The main advantage of hls4ml is that it simplifies FPGA programming, enabling real-time deployment of machine learning models without requiring deep FPGA expertise. Originally used for simple networks in high-energy physics, the paper adapts the framework for CNNs suited to image tasks like semantic segmentation.

The CNN architecture is tailored for pixel-wise classification, optimized for the limited memory and computational power of FPGAs. Techniques like pruning (removing redundant neurons) and quantization (lowering parameter precision) reduce model size and resource usage without sacrificing accuracy. Hardware-aware design further ensures the model runs efficiently on FPGA hardware.

The paper addresses the challenge of balancing accuracy with hardware constraints, aiming for optimal performance within FPGA limits. Results show the FPGA implementation offers low-latency, high-throughput inference, ideal for real-time applications like autonomous driving, accurately segmenting road scenes.

A key contribution is demonstrating how FPGAs improve energy efficiency and latency compared to CPUs and GPUs. While CPUs have higher latency and GPUs consume more power, FPGAs, customized for specific tasks, reduce power consumption and boost processing speed. [28]

This paper likely presents a method for using a FPGA to detect cracks in real-time. The paper might discuss how a particle filter, a probabilistic sequential Monte Carlo method, is used to track and detect cracks in images or videos. It could explain how the particle filter algorithm is optimized and implemented on an FPGA for real-time processing. The paper may also describe the specific features or characteristics used to identify cracks in the images or videos, such as changes in texture, intensity, or edge information. Finally, the authors will likely present experimental results demonstrating the accuracy, speed, and efficiency of the FPGA-based crack detection system. This research could have practical applications in various fields, such as infrastructure inspection, manufacturing quality control, and autonomous systems. [29]

3.3 UNet Architecture in Crack Segmentation

The U-Net architecture has proven to be highly efficient in crack segmentation, especially in the detection and measurement of road cracks. Originally developed for medical image segmentation, U-Net has been adapted to handle the challenges of segmenting road surfaces, which often involve complex and noisy environments. The architecture's encoder-decoder structure is particularly well-suited for this task as it allows for detailed pixel-wise predictions, making it capable of distinguishing cracks from the surrounding pavement with greater precision, as mentioned in the cited papers which will be studied as similar application of this thesis.

One of the primary advantages of U-Net is its ability to address the challenges posed by heterogeneous road surfaces. These surfaces often have inconsistent lighting, low contrast, shadows, and other noise that can interfere with traditional computer vision methods. However, U-Net, through its deep learning capabilities, is able to learn these features and generalize well, improving the accuracy of crack segmentation even in difficult conditions. Moreover, the segmentation of crack width is a crucial task for road maintenance, as the severity of cracks is often determined by their width. The U-Net model described in this paper excels in detecting the thickness of cracks, offering a significant improvement over previous approaches that focused only on the presence of cracks.

The paper compares the performance of this U-Net-based model with other models such as those by Lau, Nguyen, and Yu. The U-Net architecture used here showed better results, especially in terms of precision and Intersection over Union (IoU), two key metrics in image segmentation. The precision of 0.85 indicates that the model produces fewer false positives compared to others, which is important for ensuring that road management systems prioritize actual cracks. The IoU score of 0.6248 further reflects the model's accuracy in producing segmentations that closely match the ground truth, which is essential for estimating crack width and developing appropriate maintenance strategies.[30]

Data augmentation played a vital role in enhancing the model's performance. By artificially increasing the size of the Crack500 dataset, the study was able to train the U-Net model on a more diverse set of images, enabling it to generalize better and perform accurately on unseen data. This step proved particularly effective in helping the model handle various lighting conditions and surface textures, which are common in road environments.

In terms of training, the U-Net model was implemented with a ResNet50 encoder pretrained on the ImageNet dataset. This use of a pretrained encoder allowed the model to converge more quickly, with only a small gap between the training and validation accuracies, indicating that the model was learning effectively and generalizing well. The model was trained using the Adam optimizer with a learning rate of 0.0001 and achieved convergence within 20 epochs.

However, one limitation of the U-Net model may not be ideal for real-time applications where faster segmentation is required. Despite this, the model remains highly practical for use in pavement management systems, where it can be employed to process large sets of images in a non-real-time setting.

The results of the U-Net model on the Crack500 dataset were promising. It was able to accurately segment different types of road cracks, particularly those with high and moderate severity levels. When applied to real-world images, the model outperformed manual labeling efforts, especially in its ability to predict crack width with greater accuracy. This is an important factor in road maintenance, where the severity of cracks is often determined by their width, linear extent, and overall area. The U-Net model's capability to segment these dimensions more accurately than other methods highlights its potential for practical application in road condition monitoring.

In summary, the U-Net architecture is highly efficient for crack segmentation. Its ability to precisely detect and measure crack width, along with its robustness in handling complex road environments, makes it a valuable tool for road maintenance planning. Despite its slower inference speed, its high accuracy and effectiveness in both segmentation and width estimation give it a significant edge over traditional methods and other deep learning models. [31] [32]

3.4 Quantization

A very common type of quantization is by reducing the precision of weights and activations from floating-point representations to fixed-point formats, which significantly lowers memory and computational requirements while maintaining adequate accuracy. The following example is from a paper focuses on improving the efficiency of CNNs through post-training quantization, specifically for embedded platforms. Bitwidth (BW), the number of bits used to represent a number, is crucial in CNN quantization. Reducing BW from 32-bit floating-point to lower values like 8-bit, 4-bit, or 2-bit saves memory and computational resources. Fixed-point representation, with a sign bit (S), integer part (I), and fractional part (F), enables faster hardware computation due to simpler integer operations compared to floating-point.

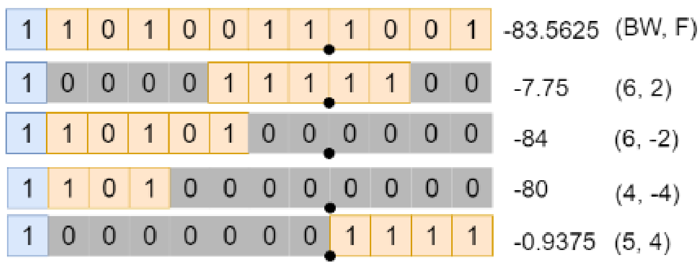


FIGURE 3.4: Examples of fixed-point representations for a given number. Bits stored in memory are in yellow and blue. Their floating-point equivalent value and corresponding BW and F are given to the right.

https://arxiv.org/pdf/2102.02147

The paper concludes that post-training quantization, especially layer-wise, minimizes accuracy loss at low bitwidths (e.g., 8-bit). It proposes an efficient quantization scheme, accounting for varying layer sensitivities, especially in the first and last layers, ensuring minimal performance degradation. Fixed-point arithmetic enhances computational speed, and per-layer quantization allows for more aggressive quantization in some layers without sacrificing accuracy. This approach reduces memory usage and computation, making quantization ideal for CNNs in embedded systems, improving inference time and preserving model accuracy. [33]

Also the paper titled "MCUNet: Tiny Deep Learning on IoT Devices" [34] addresses the challenge of deploying deep learning models on resource-constrained microcontrollers, which typically have limited memory and computational power. The authors propose a solution called MCUNet, a framework that co-designs a neural network architecture called TinyNAS and a memory-efficient inference engine known as TinyEngine.

TinyNAS automatically designs compact neural network architectures tailored for the limited resources of MCUs, while TinyEngine optimizes memory usage and execution efficiency during inference. Together, they enable state-of-the-art deep learning models to run on extremely low-power IoT devices with as little as 1MB of Flash memory and 320KB of SRAM.

The paper demonstrates the effectiveness of MCUNet by achieving significantly better performance and efficiency on MCUs compared to existing methods. The authors show that MCUNet can run image classification models with higher accuracy and lower latency, bringing deep learning capabilities to tiny edge devices that were previously unable to handle such tasks. This opens up new possibilities for real-time, intelligent applications in resource-limited environments.

Chapter 4

Modeling

4.1 General information of CNN

This chapter investigates the structure and functionality of a convolutional neural network (CNN) designed for the identification and segmentation of cracks in coastal concrete structures such as harbors and piers. CNNs have been selected due to their proven efficacy in processing image data, which aligns with the two-dimensional matrix nature of the input. The input to the neural network is a three-dimensional matrix with dimensions 3×128×128, corresponding to a 128x128 pixel image decomposed into RGB values. The output is also a three-dimensional matrix of the same dimensions, producing another RGB image. In this output image, the concrete and cracks are distinguished using high-contrast colors: yellow for cracks and purple for intact concrete. Thus, the CNN not only detects cracks but also performs their segmentation. An analysis follows, detailing the operational steps from image input to segmented output. Initially, the neural network applies several layers of downsampling to the input, followed by an equal number of upsampling layers. Concatenate layers strategically connect corresponding downsampling and upsampling layers, which share the same number of filters and matrix dimensions, facilitating information flow throughout the network. This architecture is known as a U-Net model due to its visual resemblance to the letter "U", as illustrated in the accompanying diagram.

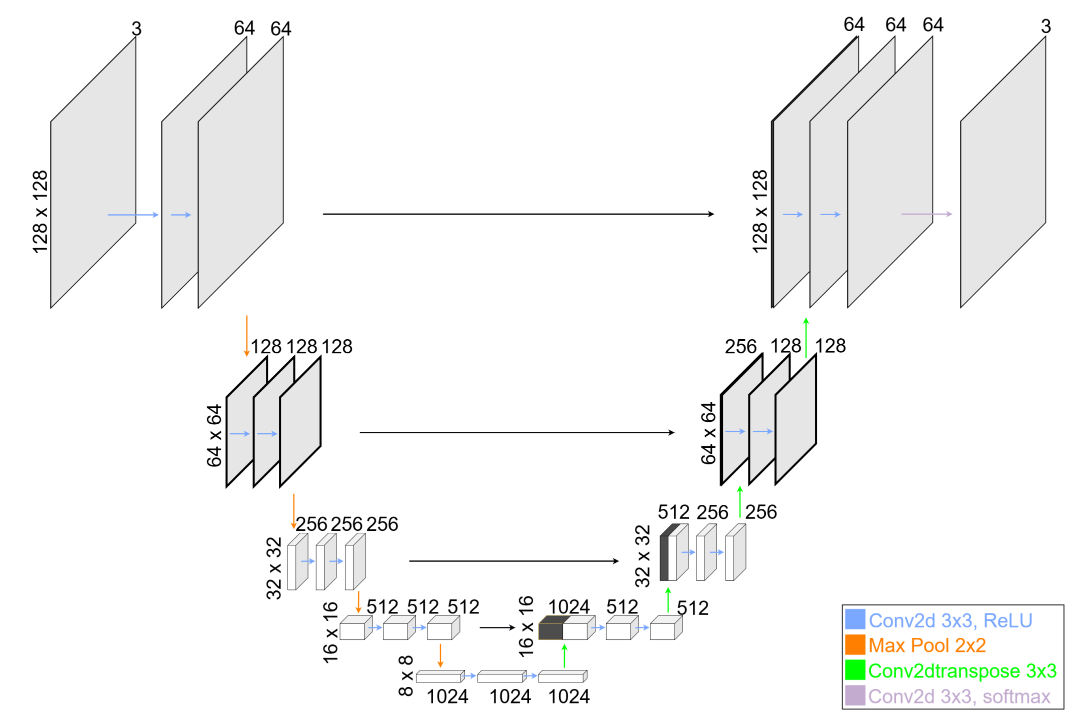


FIGURE 4.1: Convolutional neural network model for which the thesis was made

The U-Net architecture is a popular selection when it comes to the crack segmentation task due to its inherent advantages in both accurate localization and efficient data utilization, outperforming alternative architectures in these aspects. This superior performance is attributed to its distinctive design, which facilitates efficient feature learning and information preservation. The hierarchical structure of the U-Net, comprising downsampling and upsampling stages, enables the learning of features at varying scales. This multi-scale feature extraction allows the network to capture both global context and fine-grained details, which are crucial for precise crack localization. Specifically, the downsampling path, through the reduction of spatial dimensions and the concurrent increase in feature maps, facilitates the extraction of a richer set of features and the learning of abstract representations of the input image. This enables the detection of intricate patterns and structures within the image, crucial for identifying cracks. In contrast, during upsampling, the network increases spatial dimensions while integrating information from the encoded features and, potentially, from earlier layers through skip connections. By directly transferring features from the downsampling path to the upsampling path, the network mitigates the loss of fine details that can occur during the upsampling process.

4.2 Dataset used

Besides structural and functional considerations, the CNN requires specific input data preparation to ensure compatibility and optimize performance. In the absence of datasets from actual naval missions from the project's submarine, pre-existing datasets are utilized for training and testing. Specifically, the [name of the chiner that made it] dataset is employed in this study. To adapt this dataset for the CNN's requirements, several transformations are applied. Initially, a subset of 7,000 images and their corresponding masks is randomly extracted from the original dataset, which contains approximately 22,600 both images and masks. This subset should be sufficient for effective model training and testing. First, each image is decoded into three distinct arrays representing the red, green, and blue color channels using Python's $tf.io.decode\_jpeg$ function. Simultaneously, each mask is converted into a single-channel format, where each pixel is assigned a value of 1 for foreground cracks or 0 for background intact cement. This binary representation simplifies the task of distinguishing between crack and non-crack regions during the training process.

After the subset of the dataset was made three more transformation are yet to be made for a complete new dataset that will be used for training and evaluation of the model

4.2.1 Resizing

In the initial preprocessing stage, each image is resized from its original 448x448 dimensions to a standardized 128x128 resolution. While resizing may introduce some loss of information, the impact on overall accuracy in this specific project is deemed negligible. The primary advantage of image resizing is the reduction of computational burden. Smaller images require less memory and processing power, leading to significant acceleration of both training and testing phases. This is especially beneficial when dealing with extensive datasets, as is the case in this study. Moreover, resizing contributes to prevent overfitting as much as possible. By reducing the level of detail, the model is less prone to learning noise or irrelevant features present in the training data. Smaller images simplify the input, so the model can focus on capturing generalizable patterns and meaningful features that are related to the task of crack identification and segmentation. Therefore, while acknowledging the potential trade-off between image resolution and information content, the benefits of resizing outweigh the potential drawbacks, particularly in the model's overall effectiveness.

4.2.2 Augmentation

Augmentation, when applied to a dataset, introduces alterations to images that can encompass geometric transformations (e.g., rotations, flips) or adjustments in color (e.g., brightness, contrast), or a combination of both. The goal is to generate modified images while preserving the essence of the original. This technique serves a pivotal role in enhancing model performance by artificially expanding the dataset. Overfitting occurs when your model learns to perform well on the training data but fails to generalize to new, unseen data Augmentation serves as a countermeasure against overfitting by subjecting the model to a broader spectrum of data variations. Also by training on a more diversified dataset, models learn to recognize the underlying patterns and features in the images, rather than memorize of specific instances. Consequently, this leads to the development of models capable of generalizing proficiently to images it has not seen before. In the specific context of this model, only the "flip right" function is employed, with a probability of occurrence set at 50% which means that for every two images in the dataset, one will be randomly selected for augmentation.

4.2.3 Normalize

Image normalization is a crucial preprocessing step in this CNN. In the raw images, each pixel's color is represented by values ranging from 0 to 255. Normalization scales these values into a standardized range of 0 to 1. This transformation yields several benefits, contributing to the overall performance and efficiency of the model. Firstly, normalization accelerates the convergence of the training process. By scaling the input values to a common range, the network can learn more efficiently, allowing for faster updates of weights and biases. This expedited learning process reduces the computational time required for training, a significant advantage when dealing with large datasets. Secondly, normalization helps to avoid saturation of neurons in the early layers of the CNN. Without normalization, large pixel values could overwhelm these neurons, causing them to output values close to their maximum limit. This can slow down learning or even stop it altogether. Normalization ensures that all pixel values are treated equally during the learning process. Without it, pixels with larger values might dominate the learning process, leading to biased models.

After normalization which is the last one of the transformations a new dataset is generated. This new dataset is used for train both the training and validation phases of the CNN model ensuring the best learning process.

4.3 Structure

This chapter is about the structure of the neural network employed, dissecting its composition, the role of individual layers, and the logic behind design choices. The network has 40 layers, including input, convolutional, transposed convolutional, max-pooling, dropout, and concatenation layers. For a simpler representation and construction the double conv block, the downsample block and the upsample block were created which are specific layers grouped together. In the subsequent sections, a thorough analysis of each block and the architechtural model of the CNN are presented.

4.3.1 Blocks

Double conv block

The U-Net architecture often utilizes a double convolution pattern, where two convolutional layers are applied sequentially within a block. This approach offers several advantages:

Stacking two convolutional layers enhances feature learning by enabling the network to detect increasingly complex patterns, such as shapes or objects. The first layer captures basic features like edges, while the second builds on these to recognize higher-level structures. This wider context is especially useful in tasks like semantic segmentation. The double convolution strikes a balance between complexity and efficiency, offering enough learning capacity without excessive computational cost. This approach, commonly used in architectures like U-Net, is a well-established practice.

Convolution2d

Regarding the convolution operation, padding is applied to the input (e.g., a 128x128x3 image might be padded to 130x130x3) to maintain the output's spatial dimensions. This allows the convolution to cover the entire image, including the edges. The convolution weights are organized into 3x3 kernels, which slide over the input image to detect features. These kernels are drawn from the weight array, with the first nine elements forming the first kernel, the next nine forming the second, and so on. Each kernel convolves with a specific 3x3 region of the input image. For example, the first kernel processes the top-left corner, while subsequent kernels shift and convolve over adjacent regions. The results from each convolution are summed to produce a single value, which becomes part of the output feature map. After processing the first region, the 3x3 kernel shifts one step to the right and repeats the process, continuing across the input until the entire feature map is generated. This process is repeated for multiple sets of kernels, with each set producing a different output feature map. If there are N input filters and M output filters, N kernels are used to generate each of the M output feature maps. The network thus builds multiple feature maps by applying different kernels, enabling it to capture more complex and abstract patterns as it progresses through the input. Padding preserves the spatial dimensions of the output, while the convolution, shifting, and summing steps allow the network to detect features across the input image. This enables the model to learn progressively richer representations, from basic edges to complex shapes.

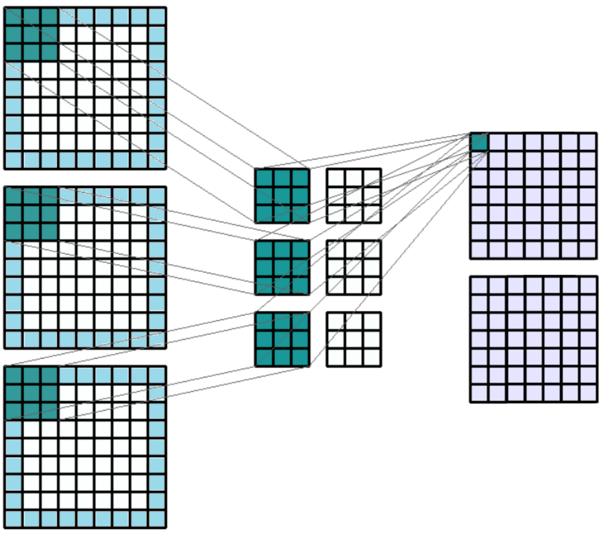


FIGURE 4.2: Convolution 2d visual
https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148

Based on the description above in this image the input is a 7x7 image with 3 channels (input filters), the output is 7x7 with 2 channels (output filters). As it can be confirmed the kernels are calculated from the multiplication of the input filters with the output filters.

$$NumOfKenrles = InFilters \cdot OutFilters = 2 \cdot 3 = 6$$

Convolutional Neural Networks (CNNs) use two types of 2D convolution, differentiated by their activation functions: one is used throughout the network, while the other, often with softmax, is applied in the final layer to generate probability distributions. The pseudocode below outlines the algorithm for a typical 2D convolution layer in CNNs:

Algorithm 1 Convolution 2D Algorithm ReLU Version

Require: $outfilters, n\_inputfilters, size, InputMatrix$
    $padded\_size = size + 2$
    $PaddedMatrix = ApplyPadding(InputMatrix)$
    for $outfilter$ **do**
        for $padded\_size - kernel\_size$ **do**
            for $padded\_size - kernel\_size$ **do**
                $element = 0$
                for $n\_inputfilter$ **do**
                    for $kernel\_row$ **do**
                        for $kernel\_col$ **do**
                            *element+=PaddedMatrix[x]**
                            *Kernel[n_inputfilter][outfilters][3*kernel_row + kernel_col]*
                $element+ = Bias[outfilter]$
                $OutputMatrix[y] = ReLU(element)$

The ReLU function can be replaced with the following equation:

InputMatrix[k] = (element > 0.0f) ? element : 0.0f

Additionally, the x, y marks are equations that utilise the variables from all the forloops for the accurate selection and placement of the element, due to the use of one-dimensional matrices. The use of one dimensional matrices and not higher dimension is explained to chapter 5 the FPGA Implementation.The second type of convolution2d is the same with a slight difference:

Algorithm 2 Convolution 2D Algorithm Softmax Version

Require: $outfilters, n\_inputfilters, size, InputMatrix$

$padded\_size = size + 2$

$PaddedMatrix = ApplyPadding(InputMatrix)$

for $outfilter$ **do**

  for $padded\_size - kernel\_size$ **do**

    for $padded\_size - kernel\_size$ **do**

      $element = 0$

      for $n\_inputfilter$ **do**

        for $kernel\_row$ **do**

          for $kernel\_col$ **do**

            *element+=PaddedMatrix[x]**

            *Kernel[n_inputfilter][outfilters][3*kernel_row + kernel_col]*

    $element+ = Bias[outfilter]$

    $OutputMatrix[y] = element$

$Softmax(OutputMatrix)$

As it can be observed the activation function change position as the softmax can not be applied until the whole matrix is completed. All the other lines are still the same with the previous algorithm. Below is the softmax function implemented in C.

Algorithm 3 Softmax Algorithm

Require: $filters, size$

for $size$ **do**

  for $size$ **do**

    $max\_val = -infinite$

    for $filters$ **do**

      if $OutputMatrix[x] > max\_val$ **then**

        $max\_val = OutputMatrix[k]$

$exp\_sum = 0$

for $filters$ **do**

  $OutputMatrix[x] = expf(OutputMatrix[x] - max\_val)$

  $exp\_sum+ = OutputMatrix[x];$

for $filters$ **do**

  $OutputMatrix[x]/ = exp\_sum;$

The apply padding function is used to apply a layer of padding to each channel of the input images. For example, if the matrix k*k*n is given as input, the output will be (k+2)*(k+2)*n. The algorithm used to achieve this is as follows:

Algorithm 4 Padding for same dimensions Algorithm

Require: $n\_filters$, $size$, $padded\_size$
  for $n\_filters$ **do**
    for *padded_size_i* **do**
      for *padded_size_j* **do**
        if *(i==0 | | j==0 | | i==size+1 | | j==size+1)* **then**
          $PaddedMatrix[k] = 0$
        else
          $PaddedMatrix[k] = InputMatrix[l]$

It would appear that the PaddedMatrix is essentially similar to the InputMatrix, but with the addition of two rows of zeros, one at the top and one at the bottom, and two columns of zeros, one on the left and one on the right.

The keras conv2d function has more parameters which were not mentioned during initialization such as stride, they have been set to their default values for this model.

4.3.2 Downsample block

The downsample block is essential in image processing, because it carries out the pixel compression and feature extraction. Each filter, represented as a 2D matrix (image), undergoes a halving of its dimensions within this block. The process starts off with the previously analyzed double convolution block, followed by the the max pooling layer, the core of downsampling.

Maxpooling layer

Maxpooling is a widely-used technique in image processing that excels in several key areas, for example it excels at edge/boundary preservation, as it tends to retain the most salient features within a region, cracks are often characterized by thin, elongated structures with high intensity variations compared to the surrounding background. Max pooling excels at capturing these strong activations within a pooling region, effectively highlighting the presence of cracks. Additionally, by focusing on the maximum values, MaxPooling helps reduce the impact of noise or minor variations in the image, leading to more robust segmentation results.

Algorithm 5 Maxpooling

Require: $depth, size$
  $output\_size = size/2$
  for $depth$ **do**
    for $size$ **do**
      $max = InputMatrix[m]$
      for $size$ **do**
        for $two1$ **do**
          for $two2$ **do**
            if $InputMatrix[k] > max$ **then**
              $max\_val = InputMatrix[k]$
        $OutputMatrix[l] = max$
  for $size/4$ **do**
    $InputMatrix[n] = OutputMatrix[n]$

Dropout layer

The final layer in the downsample block is a dropout layer, with a probability of weight ignorance (Pi) set to 0.3. This layer randomly deactivates 30% of neurons, promoting network resilience and preventing overfitting. It is important to note that this dropout layer is only active during the training phase and is not applied during inference or testing, reducing the downsampling block to three layers: the double convolution and maxpooling layers.

Overall, the downsample block is a fundamental component in image processing, responsible for reducing image dimensions, extracting meaningful features, and improving model generalization and performance. The combination of double convolution, maxpooling, and dropout regularization makes this block a powerful tool in various image-related tasks, such as segmentation and classification.

4.3.3 Upsample block

The upsample block is equally important in image processing, because basically it recreates the image with the features extracted from the previous downsample blocks. This block carries out the dimensionality increase and it consists of five layers.

Convolution 2d Transpose

The first and most significant layer in this block is the Conv2dTranspose. This layer is crucial when upsampling is needed, as it reverses the process of standard convolution, restoring the input shape while preserving element relationships. It is often preferred over methods like UpSampling2D, particularly for tasks like crack segmentation, where detailed and accurate upsampling is required to capture fine features. Conv2dTranspose excels by learning specialized upsampling filters, producing more precise segmentation masks for crack detection. While similar to Conv2d in computation, Conv2dTranspose differs mainly in padding and lacks an activation function, with a larger input matrix size.

Algorithm 6 Convolution 2D Transpose

Require: $outfilters, n\_inputfilters, size$
  $padded\_size = 2 * size + 1$
  $PaddedMatrix = ApplySpecialPadding(InputMatrix)$
  for $outfilter$ **do**
    for $padded\_size - kernel\_size$ **do**
      for $padded\_size - kernel\_size$ **do**
        $element = 0$
        for $n\_inputfilter$ **do**
          for $kernel\_row$ **do**
            for $kernel\_col$ **do**
              *element+=PaddedMatrix[x]**
              *Kernel[n_inputfilter][outfilters][3*kernel_row + kernel_col]*
        $element+ = Bias[outfilter]$
        $OutputMatrix[k] = element$

In the convolution transposed, the padding is somewhat more intricate, comprising two distinct phases. Initially, each row and column is augmented with an S-1 number of zeros, in this particular instance equating to zeros = 2 - 1 = 1. This results in the matrix assuming the following configuration:

Additionally, the second step is to solve the equation below to determine the external padding of the matrix. This involves replacing the input (i), kernel (k), and stride (s) values, as well as the size of the output feature map (o), with the known values in order to ascertain the padding (p).

$$o = (i - 1) * 2 + kernel\_size + 2p$$

In this particular instance, the CNN model exhibits four convolution2d transposes, where in the p value is 0.5. This indicates that the padding should be partial, necessitating the addition of a row of zeroes solely on the right side of the matrix and a column on the bottom. Upon completion of this padding process, the convolution computation shall ensue.

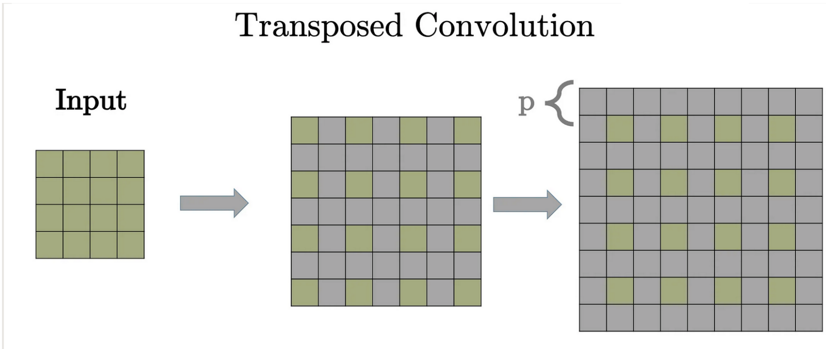


FIGURE 4.3: Convolution transpose 2d visual
https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148

This logic is embodied to the following algorithm.

Algorithm 7 ConvTranpose padding

Require: $n\_inputfilters, size, padded\_size$
  for $n\_filters$ **do**
    for $padded\_size$ **do**
      for $padded\_size$ **do**
        $padded\_index = k$
        if
          then$positional\_condition$
          $PaddedMatrix[k] = InputMatrix[l]$
        else
          $PaddedMatrix[k] = 0$

Concatenate

Concatenate layers in a U-Net model merge feature maps from the downsampling and upsampling paths, allowing the decoder to access both high-level semantic information and fine spatial details. These layers also help prevent vanishing or exploding gradients by creating skip connections that ensure stable training. Additionally, they assist in recovering spatial resolution lost during downsampling, resulting in more precise segmentations. Below is the algorithm

Algorithm 8 Concatenate algorithm

```
for Depth do
    for size do
        for size do
            TempMatrix[concatIndexA] = InputMatrix[concatIndexA]
            TempMatrix[concatIndexB] = CombinedMatrix[concatIndexB]
for size * size * depth * 2 do
    OutputMatrix[ka] = TempMatrix[ka]
```

Dropout

The upsampling block concludes with a dropout layer followed by a double convolution block. These layers serve identical purposes to their counterparts in the downsampling block, so there is no the need for further analysis.

4.3.4 UNet Architechture

This subsection describes the architecture of the crack segmentation model, utilizing the building blocks mentioned before and additional layers. The model follows the classic U-Net model, renowned for its efficacy in image-to-image tasks. Its components are presented below:

Input Layer

The definition of the input for u-net model. Typically tf.keras.Input is used at the very beginning of the model definition, specifying the shape of your input data.

Encoder

The encoder section of the model consists of four sequential downsample blocks. Each block receives an input, produces two identical outputs, and progressively reduces the image dimensions while doubling the number of convolutional filters. The first output is passed to the next downsample block, while the second is sent to a corresponding concatenate layer in the decoder for skip connections. The alias of those blocks is encoder because those blocks progressively narrowing the information flow while retaining the most important details.

Bottleneck

The bottleneck, receiving the output of the final downsample block, applies a double convolution with 1024 filters. This layer acts as a bridge between the encoder and decoder. It is called bottleneck because upon observing the unet model it is the narrowest point of the model visually resembling the neck of a bottle.

Decoder

The decoder, mirroring the encoder, consists of four upsample blocks, each receiving input from both the corresponding downsample block (via skip connections) and the prior upsampling layer. Functionally, the decoder decodes or reconstructs the compressed information from the encoder, progressively doubling image dimensions while halving the number of filters. This allows the network to recover spatial details and generate a refined segmentation mask.

Output block

The last layer, acting as the output layer, is a slightly different type of convolution compared to the ones used earlier in the U-Net. This layer uses a default weight initialization method called Glorot Uniform, which sets the initial values of the 3x3 kernel (filter) using a special formula. This formula calculates a spread of values based on the number of connections going into and out of the layer. This helps keep the signals passing through the network stable. After the convolution, the sigmoid function is applied. This function converts the output of the convolution into a probability between 0 and 1. This is important for crack detection because the goal is to predict whether each pixel in the image is part of a crack (closer to 1) or not (closer to 0). Also the output layer reduces the number of filters from 64 to 3 so the output is three images of 128 x 128 pixels. In these images the cracks will be captured by the unet model.

4.4 FPGA design issues

Introduction about the how many storage does the wieghts use of the network, exploring option to reduce the storage the wieghts and the biases take so that all can fit in a bram and the cnn accelerator be actually faster than the original. that comes in exchange of some of the accuracy, but the speed and the accuracy relation ship is not 1:1 so the model can gain an big speed boost without losing too much on its performance. There are two methods which were considered for this accomplishment, the first is a framework developed from the Politechinco di Torino and the second is the k-means technique.

In the original cnn there are 34,5 million parameters, each one a 32 bit float, so the total capacity needed to store them is calculated like at 131,66 MB. Each float takes up 4 bytes of memory so multiplying the number of parameters by the bytes per float gives you the total memory usage in bytes:

$$34,513,475(param) \cdot 4(bytes/param) = 138,053,900(bytes)$$

Having the number of bytes a conversion to megabytes is applied

$$138,053,900(bytes)/1,048,576(bytes/MB) \approx 131.66(MB)$$

The onboard bram in the Alveo U55C fpga is at 70.9 MB, meaning that exernal hbm should be used if it for the wieghts to be used as they are. So the design would not be as good at accelerating the original beacuse there would be a limitation at pipelining the weights.

4.4.1 Polito's framework

The paper presents a new methodology for designing efficient hardware accelerators mainly for ResNets on FPGAs. The authors [35] address this challenge by introducing an optimized architecture for CNNs that minimizes the buffering resources required by residual blocks. This optimization is crucial for enabling on-chip storage of model parameters and activations using 8-bit quantization, which strikes a balance between accuracy and resource utilization. The proposed architecture is implemented using a custom HLS flow, which translates the Python model description into C++ code suitable for FPGA synthesis. The key innovation of the work lies in the efficient management of skip connections within residual blocks. The authors propose a combination of graph optimization techniques, including loop merging and temporal reuse, to reduce the buffering overhead associated with skip connections. This optimization not only conserves on-chip memory but also ensures smooth data flow within the accelerator, preventing stalls and maximizing throughput. The effectiveness of the proposed architecture and design flow is demonstrated through experiments on the CIFAR-10 dataset using ResNet8 and ResNet20 models. The implementations target two FPGA boards: Ultra96-V2 and Kria KV260. The results show that the proposed approach achieves state-of-the-art performance in terms of throughput, accuracy, and energy efficiency compared to existing FPGA-based ResNet accelerators and other frameworks like Vitis AI and FINN. The paper's methodology is further clarified through a detailed analysis of the implementation steps. This approach includes a series of interconnected stages, each contributing to the optimization and realization of high-performance accelerators on FPGA platforms.

- Quantization: The process begins with the quantization of the neural network model. Quantization involves converting the model's weights and activations from floating-point representation to fixed-point representation. This is done using the Brevitas framework, which allows for quantization-aware training. The weights and activations are quantized to 8-bit integers, while biases are quantized to 16-bit integers. This quantization is chosen as a balance between model accuracy and hardware efficiency.
- Code Generation: Once the model is quantized, the next step is to generate C++ code for the FPGA implementation. This code generation process involves extracting the network graph in the QONNX format, which provides a structured representation of the network's layers and connections. The generated C++ code includes a top-level function that instantiates the tasks required for network inference, such as computation tasks for convolutional and pooling layers, parameter tasks for

managing convolution parameters, and window buffer tasks for formatting input data.

- High-Level Synthesis: The generated C++ code is then synthesized into RTL code using Vitis HLS, a high-level synthesis tool from Xilinx. The synthesis process leverages a set of model-independent C++ libraries that implement optimized layer operations. These libraries are designed to be reusable and adaptable to different tensor dimensions, data types, and levels of parallelism.
- Bitstream Generation: The final step involves importing the generated RTL code into the Vivado Design Suite, another tool from Xilinx. Vivado is used to implement the design on the target FPGA and generate the bitstream, which is the configuration file used to program the FPGA.

The methodology's effectiveness is further amplified through a series of optimizations. Pipelining, employed at both the inter-task and intra-task levels to enhance throughput. Inter-task pipelining enables simultaneous execution of different layers, while intra-task pipelining allows for simultaneous execution of operations within a layer. Additionally, the design utilizes DSP packing, a technique that allows multiple multiply-accumulate (MAC) operations to be performed on a single DSP block in the FPGA. This optimization is particularly effective for quantized data and significantly reduces the hardware overhead of computations. Throughput is further enhanced by employing an integer linear programming (ILP) model to optimize loop unrolling within computation tasks, effectively maximizing throughput under the constraints of limited DSP resources. The design also incorporates efficient window generation to minimize on-chip memory requirements for activations, ensuring a streamlined data flow. Finally, the methodology includes graph optimization techniques specifically tailored for residual networks. These optimizations aim to reduce the buffering overhead associated with skip connections, which are a key feature of ResNets. By minimizing the buffering requirements, the design achieves better resource utilization and higher throughput.

4.4.2 Why not used

Despite this framework's product being so efficient and have high accuracy/ quantization ratio there are three reasons that it can't be used to reduce the size of this CNN. Firstly, the paper provided focuses on optimizing ResNets, which are a specific type of CN architecture. UNet, while also a CNN, has a distinct architecture designed for image segmentation tasks. Applying the optimizations discussed in the paper to a UNet model would have varying effects which can be seperated to optimizations which can be applied and to optimization that are non-applicable. The quantization, the hls implementation are the optimization that can be applied to all the CNNs despite being ResNet or something else. Also The pipelining and DSP packing optimizations used to improve throughput in ResNets could also be applied to the convolutional layers in a UNet model. But the skip connection management which is the core of this framework can not be utilized. UNet also has skip connections, but they serve a different purpose (combining spatial information at different scales) and have a different structure than those in ResNets. The specific optimizations for ResNet skip connections would not directly apply to UNet.

The second problem occured with the quantization of the model. As mentioned above the Framework requires the input to be at qonnx type, which means while exporting the model in keras a simple conversion can turn the type of the model at onnx. From here when a type of quantization is applied the model will be qonnx, this framework works with a specific type of quantization the brevtias which is a PyTorch library specifically designed for neural network quantization made from amd xilinx. As mentioned in the Polito's paper the type of the brevitas qunatization is

$$\alpha = Q(b) = clip(round(b \cdot 2^{b_w - s}), \alpha_{min}, \alpha_{max}) \cdot 2^8$$

where:

- α is the quantized value.
- Q(b) is the quantization function applied to the original value b.
- $clip(x, \alpha_{min}, \alpha_{max})$ clamps the value x between α_{min} and α_{max}.
- $round(x)$ rounds the value x to the nearest integer.
- b_w is the number of bits used for quantization.
- s is a scaling factor.

The clipping bounds α_{min} and α_{max} are determined based on whether the quantization is signed or unsigned:

$$a_{min} = act_{min}(s) = \begin{cases} 0 & \text{if unsigned} \\ \\ -2^{b_w-1-s} & \text{if signed} \end{cases}$$

$$a_{max} = act_{max}(s) = \begin{cases} 2^{b_w-s}-1 & \text{if unsigned} \\ \\ 2^{b_w-1-s} & \text{if signed} \end{cases}$$

Brevitas quantization can be applied via two methods, the quantization aware training (QAT) which is applied during the creation of the model and when the training is done the quantization is applied to the weights. and the second method is the post training quantization (PTQ) which is after the model is trained and the weights are stable then the quantization is applied to make the reduction. In this case of the U-net only the second option was available because the initial model should remain unchanged and was already done so there was an efficient option time-wise. The tool has one disadvantage, the PTQ option accepts only a handfull of torchvision models for quantization and not custom inputs so there wasn't possible for it to accept the unet as it wasnt among the torchvision models. There was an attempt of reformatting the code of brevitas so that it could accept custom models too after the encouragement of the maintainer of the repository on the github, but the results of the program was not good enough to continue working with it. The last and most important reason for abandoning this option was that in this stage of the framework the option for Alveo U55C didn't exist so the FPGA used in the specific project was missing from the framework's available options. Based on these reasons the option to explore something else to make the CNN fit in the FPGA's was chosen.

4.5 Kmeans

4.5.1 introduction to k-means

So a new weight reduction technique should be used because the Polito framework wasn't not avalaible at the moment. So the second implementation to reduce the bytes of the weights is the k-means method. The k-means method is an unsupervised machine learning algorithm used for clustering. This means it is a procedure for partitioning an N-dimensional population into k sets based on a sample.The goal is to create partitions that minimize within-class variance, meaning the variance of points within each group is as small as possible. The iterative process starts by selecting k random points as initial "means" or cluster centers. Each data point is assigned to the nearest mean, and the means are recalculated based on the new assignments. This continues until assignments stabilize or a maximum number of iterations is reached. This method is computationally efficient and applicable to large datasets, used in various applications like identifying groups of similar data points (similarity grouping), predicting values based on other variables (nonlinear prediction), estimating probability distributions of multiple variables, and testing the statistical independence of multiple variables.The within-class variance, the key concept int the k-means method, is defined by the following formula:

$$W^{(2)}(S) = \sum_{i=1}^{k} \int_{S_i} ||z - u_i||^2 dp(z)$$

where

- $S = S_1, S_2, ..., S_k$ is a partition of the data into k sets.
- u_i is the conditional mean of the probability distribution p over the set S_i.
- $||z - u_i||^2$ represents the squared Euclidean distance between a data point z and the mean u_i of its assigned cluster.

The minimization of within-class variance in the k-means algorithm is achieved through a systematic process. Initially, k points are randomly selected from the dataset to serve as the initial cluster centers. Each data point is then assigned to the nearest cluster center based on Euclidean distance. Following this, the cluster centers are recalculated as the mean of the data points assigned to each cluster. These steps of reassignment and recalculation are repeated until a stopping criterion, such as a maximum number of iterations or a threshold for change, is met. During the assignment step, each data point is assigned to the nearest cluster center. Consider a data point zz and its assigned cluster center xx. The contribution of this data point to the within-class variance of its cluster is the squared Euclidean distance between the point and the center: $||z - x||^2$. If the data point is reassigned to a new cluster center yy that is closer, the new contribution to the within-class variance becomes $||z - y||^2$. Since yy is closer to zz than xx, we have $||z - y||^2 < ||z - x||^2$. Thus, reassigning the data point to the closer center reduces its contribution to the within-class variance. When this is done for all data points, the overall within-class variance decreases. Further minimization occurs during the update step, as the cluster centers are recalculated to be the mean of the assigned points. The mean is the point that minimizes the sum of squared distances to all points in the cluster, ensuring a reduction or no change in the within-class variance. Mathematically, the new center yy that minimizes the within-class variance can be shown to be the mean of the data points in the cluster: $y = \frac{1}{n}\Sigma_{i=1}^{n} z_i$. By iteratively reassigning data points to the nearest centers and updating the centers to be the means, the k-means algorithm progressively reduces the within-class variance. This process continues until the algorithm converges, meaning the cluster assignments and centers stabilize. However, this solution may not always be the global optimum, as the final result can be influenced by the initial random selection of cluster centers.

4.5.2 K number selection

Selecting the optional k value in k is a crucial step, as it directly impacts the quality and interpretability of the resulting clustering. There are a number of algorithms that can be employed to identify the optimal k, including the Elbow method, Silhouette Analysis, Gap Statistic, Information Criteria and Cross-Validation. While there is no single "best" method for choosing k in k-means, an understanding of the relative strengths and weaknesses of each approach can inform the decision-making process. The Elbow method is an excellent starting point due to its simplicity and intuitive visual interpretation, particularly when the data exhibits a clear "elbow" point in the plot. However, it may not be suitable for all datasets, particularly those with smooth curves or less distinct clusters. Silhouette analysis offers a more robust evaluation of cluster separation and cohesion, providing insights into how well each data point fits within its assigned cluster. In contrast, the Gap statistic is useful when there is a suspicion about the number of clusters and a statistical validation is desired. Information Criteria, such as AIC and BIC, and Cross-Validation offer a more statistically rigorous approach, but may be less intuitive to interpret. Ultimately, the choice depends on factors such as the desired interpretability, computational resources, assumptions about cluster shapes, data characteristics, and the level of statistical rigour required. It is frequently advantageous to integrate multiple methodologies and draw upon domain expertise to make the most informed determination regarding the optimal value of 'k'. The most commonly employed approach is the elbow method, which is readily applicable without any significant computational burden.

Despite having so many approaches on how to choose k, in this study the k was choosen based on the memory efficinecy for the Alveo board. The reason memory efficiency was chosen as the deciding factor and not accuracy is because the goal of this application is the accelaration of the CNN. So the first thought was to fit into 32 bit multiple weights instead of only one. Before explaining the reasoning the board's capabilities should be known, based on the dataset the onboard bram in the Alveo U55C fpga is at 70.9 MB. As mentioned before the parameters of this CNN does not fit all together stimultaniously. With the exact number of 34,513475 parameters it is approximatelly double the avalaible bram.

In determining the number of clusters (k) for k-means, a key consideration was the efficient use of Block RAM (BRAM), a shared resource in FPGAs. To avoid over-allocation of BRAM to CNN parameters, a guideline of 50% maximum BRAM usage was established as the initial guideline. The 8-bit k-means scenario was identified as the most BRAM-intensive. In this case, each weight is represented by 8 bits, allowing four weights to be stored in a 32-bit word as presetned above. This effectively reduces BRAM usage by a factor of four of what would be required for uncompressed weights. After this optimization, the 8-bit k-means approach still utilizes 46.43% of the available BRAM (32.92 MB).

32 - 24	23 - 16	15 - 8	7 - 0
4th weigth	3rd weight	2nd weight	1st weigth

With these considerations in mind, 6-bit k-means was selected as the initial configuration. This choice allows for flexibility, as the number of clusters can be adjusted based on the results, either increasing to 7-bit or 8-bit, or decreasing to 5-bit or 4-bit. The 6-bit k-means approach significantly reduces the number of parameters compared to the original model. Since each 32-bit word can accommodate five 6-bit indices (with two bits remaining unused), the parameter storage is effectively compressed by a factor of 5. This compression translates to a substantial reduction in BRAM usage, from 131.66 MB to 26.34 MB, representing 37.15% of the total BRAM capacity.

31 - 30	29 - 24	23 - 18	17 - 12	11 - 6	5 - 0
useless bits	5th weigth	4th weigth	3rd weight	2nd weight	1st weight

If the 6-bit k-means model achieves satisfactory accuracy, the next evaluation involves testing 4-bit k-means with 16 clusters (compared to the 64 clusters of the 6-bit model). This assessment aims to determine whether the reduced bit precision and fewer clusters significantly impact accuracy. Should the 4-bit k-means model maintain high accuracy, it offers a remarkable efficiency advantage. By storing eight weights in each 32-bit word, it achieves an 8:1 compression ratio compared to the original parameter representation. Consequently, BRAM usage would be reduced to a mere 16.46 MB, utilizing only 23.22% of the total available BRAM.

31 - 28	27 - 24	23 - 20	19 - 16
8th weight	7th weight	6th weight	5th weight

TABLE 4.1: 4-bits kmeans weight placement in 32 bits (Part 1)

15 - 12	11 - 8	7 - 4	3 - 0
4th weight	3rd weight	2nd weight	1st weight

TABLE 4.2: 4-bits kmeans weight placement in 32 bits (Part 2)

If the 4-bit k-means model's performance proved to be moderate to bad, also 5-bit kmeans is better from the 6-bits one, beacuse it can hoold an additional number so it would reduce the memory usage to 21.95 MB so the 30.96% usage of the bram.

31 - 30	29 - 25	24 - 20	19 - 15
useless bits	6th weight	5th weight	4th weight

TABLE 4.3: 5-bits kmeans weight placement in 32 bits (Part 1)

10 - 14	9 - 5	4 - 0
3rd weight	2nd weight	1st weight

TABLE 4.4: 5-bits kmeans weight placement in 32 bits (Part 2)

On the other hand should the 6-bit k-means model prove inadequate, a 7-bit k-means configuration with 128 clusters would be evaluated. While this configuration, like the 8-bit model, stores four weights per 32-bit word (with four unused bits), it offers a valuable additional data point for comparison.

31 - 28	27 - 21	20 - 14	13 - 7	6 - 0
useless bits	4th weight	3rd weight	2nd weight	1st weigth

Further increases in the number of clusters (k) would lead to a major rise in BRAM usage. Therefore, configurations with 512 or more clusters were deemed impractical for this network due to the excessive resource demands they would impose.

4.5.3 Metrics for evaluation

In the realm of image segmentation, evaluating the performance of algorithms is paramount to understanding their efficiency and reliability. Two widely used metrics for this purpose are the Intersection over Union (IoU) and the Dice Coefficient. IoU, also known as the Jaccard Index, measures the overlap between the predicted and ground truth segmentation masks by comparing the intersection and union of these sets. On the other hand, the Dice Coefficient, or Sørensen-Dice Index, emphasizes the similarity between the predicted and true regions by considering the harmonic mean of their sizes. While both metrics aim to quantify the accuracy of segmentation results, they offer distinct perspectives and have unique sensitivities to various segmentation challenges. These metrics were considered to be used for the comparison between the original model and the other kmeans models. Regarding the dice coefficient we have that:

$$DiceCoefficient = 2 \cdot \frac{|X \cap Y|}{|X| + |Y|}$$

where

- $|X|$ is the cardinality of the first set (the number of elements in set X)
- $|Y|$ is the cardinality of the second set (the number of elements in set Y)
- $|X \cap Y|$ is the cardinality of the intersection of the sets (the number of elements that are in both sets X and Y)

and regarding the IoU metric has the mathematical formula of:

$$IoU = \frac{|X \cap Y|}{|X \cup Y|}$$

where

- $|X|$ represents the area of the predicted region or bounding box.
- $|Y|$ represents the area of the ground truth region or bounding box.
- $|X \cap Y|$ represents the area of intersection between the predicted and ground truth regions.
- $|X \cup Y|$ represents the area of union between the predicted and ground truth regions.

To determine the most appropriate metric for evaluating crack segmentation in this context, a comparison of Intersection over Union (IoU) and Dice coefficient is essential, considering the unique challenges this task presents. IoU prioritizes boundary precision, making it ideal for applications where accurate crack delineation is critical, such as measuring crack dimensions. Its intuitive interpretation as a ratio simplifies result analysis and comparison. However, IoU can be sensitive to small or fragmented cracks, leading to unrepresentative evaluations due to significant metric reductions. This limitation is particularly relevant when dealing with datasets containing such cracks, as is the case in our scenario. Dice coefficient, conversely, demonstrates robustness to these challenges. It emphasizes overall region overlap, making it more tolerant of the thin, fragmented, or discontinuous nature of cracks. This results in a more reliable assessment of segmentation quality in real-world scenarios, especially when dealing with small cracks present in the dataset. Furthermore, its ability to handle class imbalances, common in crack segmentation, enhances its applicability. In summary, while IoU may be preferred when precise boundary localization is paramount, the Dice coefficient emerges as a more versatile and reliable metric for evaluating crack segmentation, particularly when dealing with datasets containing small or fragmented cracks. Given the presence of such cracks in our dataset and the Dice coefficient's ability to handle these common challenges, it has been chosen as the primary metric for evaluating the performance of our models.

4.5.4 Results phase

This chapter presents the results of the software test. The test dataset consists of 500 images randomly selected from the original training dataset of approximately 11,000 images. This subset offers a balance between testing speed and representative accuracy assessment. Each image is presented with its corresponding mask, followed by the model's output. The models being evaluated utilize 5-bit, 6-bit, 7-bit, and 8-bit k-means quantization. The 4-bit k-means model was excluded due to observed inaccuracies in preliminary testing. Based on the metrics and visual similarity, this analysis will determine the optimal reduced-weight model for hardware implementation. The process of the crack segmentation begins with feeding the image into the CNN, where the RGB (Red, Green, Blue) values of each pixel are analyzed. Cracks are identified based on the variations in these RGB values, which differ from the surrounding areas. These differences in color can be due to various factors such as lighting conditions, the material properties, and the nature of the crack itself. For example, in materials like asphalt, cracks might appear darker, whereas in lighter materials like white paint, cracks might be brighter. In the case of cement, cracks usually manifest as darker lines against the lighter background. Key features for crack detection are the edges and gradients within an image. Edges represent significant changes in color or intensity and are critical indicators of cracks. The CNN uses convolutional layers to detect these edges by applying various filters that highlight areas where RGB values change abruptly. Beyond just detecting edges, the CNN also calculates gradients, which measure the rate of change in color values. High gradients typically correspond to the boundaries of cracks, helping the model to more accurately identify these features. The texture of an area in an image, defined by the pattern of pixel intensities, is another crucial factor in crack detection. Cracks often have a distinct texture, characterized by a series of alternating light and dark pixels. The CNN learns to recognize these textures during training, enabling it to distinguish cracks from the smooth, unblemished areas of the material. In the analysis which was made in this study, the CNN performed well in most cases, correctly classifying images with cracks and those without. However, some challenges arose with ambiguous cases producing unclear or incomplete crack segmentations, making it difficult to confidently interpret the results. In a few instances, the CNN detected cracks in images of pristine cement. This could be due to the presence of features (like shadows or stains) that shared some characteristics with cracks. Ultimately, successful crack detection relies on how much a crack stands out from the surrounding cement in terms of color, edges, and texture. CNNs offer a powerful tool for automating this process, but it's important to be aware of their limitations and interpret results cautiously in ambiguous situations. The model's results fall within these four classifications.

True positives The model successfully detected and clearly identified cracks in some images, replicating them accurately in the output. However, in other cases, while cracks were detected, they were not fully imprinted, with only partial identification. The project's primary focus is on the model's ability to detect cracks, rather than perfectly replicating their exact shape as in the mask.

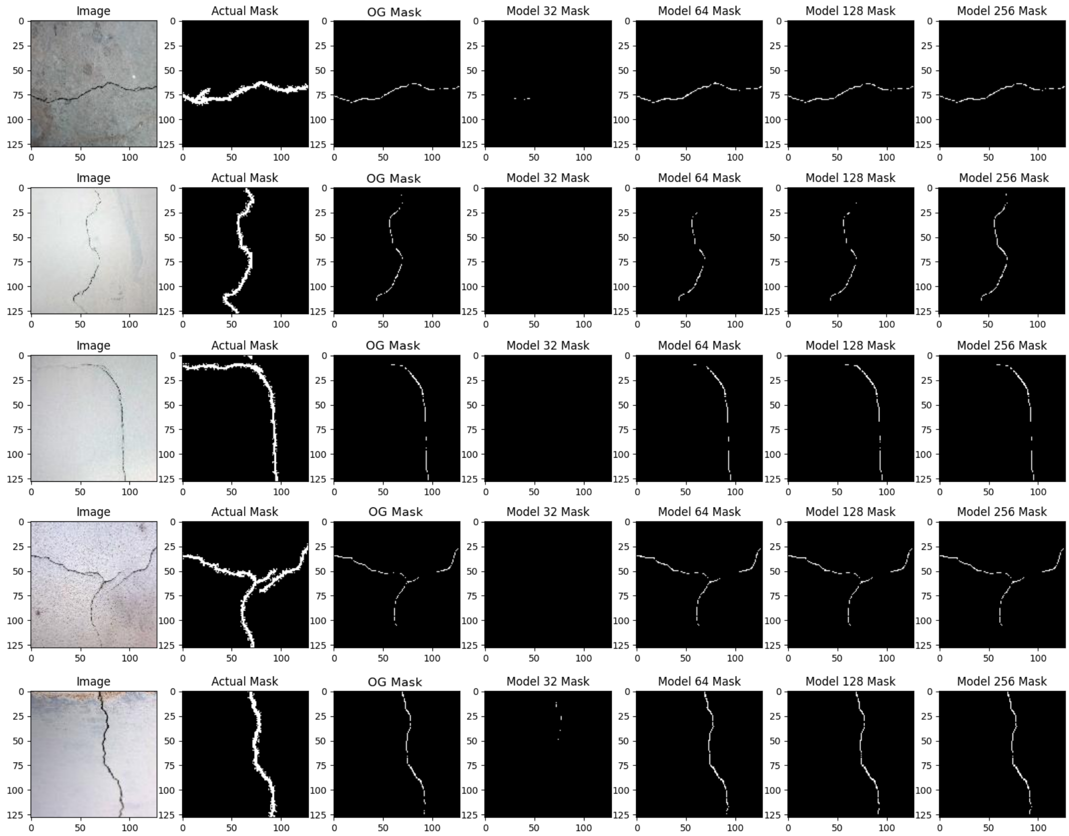


FIGURE 4.4: True positives results which found well

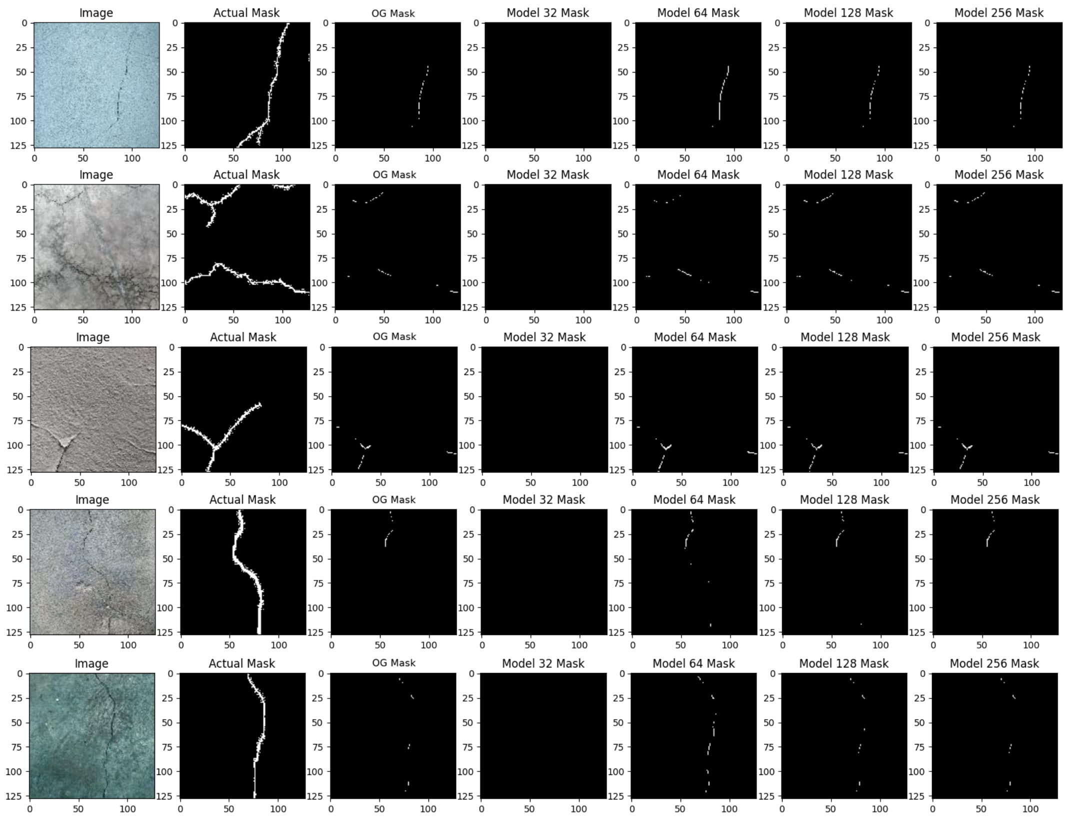


FIGURE 4.5: True positives results which slightly found

True negatives There were images which were pure cement which means that there were no cracks in those images. The model did not recognise anything as a crack so there was a correct true negative classification by the model.

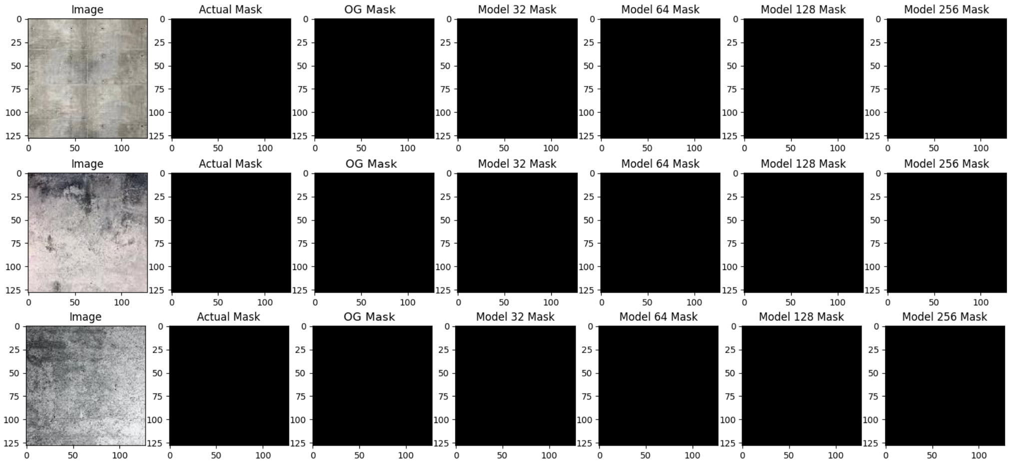


FIGURE 4.6: True negative results

False positives There were images with no cracks where cracks were found despite having no cracks whatsoever. Those images contain something with rapid colour change, as exaplained before something like this can confuse the model and make it detect a crack. For example a brick wall, a road marking or a road well. These kind of items adds to the cement picture this colour differentation that makes the model ifentify them as cracks.

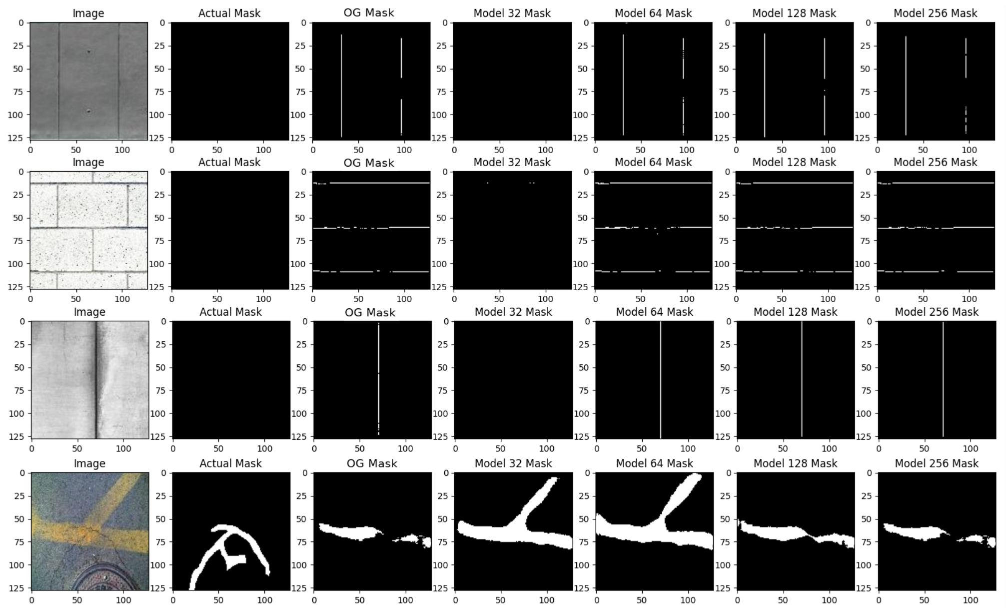


FIGURE 4.7: False positives results

False negatives Some images missed cracks because the cracks were too small to stand out, even to the naked eye. Resizing the images further reduced important details, causing the model to overlook these cracks. Unlike false positives, there were no distracting objects, the cracks were simply too narrow to be detected by the model.

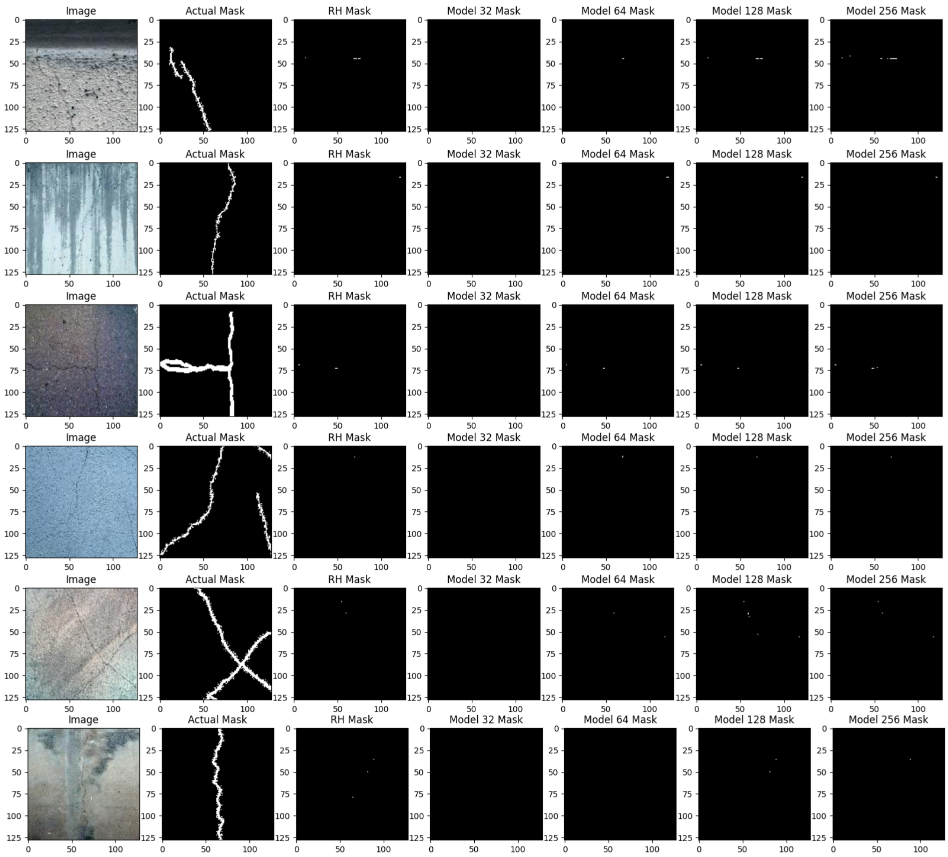


FIGURE 4.8: False negative results

By using the dice coefficient there can be a comparsion between the masks and each model.

Model	Lowest Dice	Mean Dice	Highest Dice
Original	0.00	0.21	0.85
RW 32	0.00	0.05	0.68
RW 64	0.00	0.21	0.85
RW 128	0.00	0.22	0.86
RW 256	0.00	0.22	0.85

TABLE 4.5: Overall Dice Coefficient Statistics for Various RW Values

In evaluating reduced-weight models, the 256 (8-bit) and 128 (7-bit) k-means models show identical performance with the same mean Dice coefficient for crack masks. The 64 (6-bit) model performs slightly worse but remains comparable, indicating similar segmentation accuracy. However, the 32 (5-bit) model is highly inaccurate, with a mean Dice coefficient of 0.05. All models, including the original, have a minimum Dice coefficient of 0.00, reflecting no overlap in the worst-case scenarios. The RW 128 model achieves the highest Dice coefficient of 0.86, indicating the best overlap between predicted and ground truth segmentations. The original and other RW models also show strong maximum coefficients, reaching up to 0.85. Despite a relatively low mean Dice of 0.22, influenced by false positives or undetectable cracks, the reduced models effectively mimic the original model and achieve the primary goal of crack detection, even if not perfectly matching crack shapes. Regarding crack detection accuracy, the models' performance can be summarized as follows:

Model	**Percentage of Cracks found**
Original	39.56%
RW 32	10.23%
RW 64	38.01%
RW 128	39.20%
RW 256	39.67%

TABLE 4.6: Percentage of Cracks found

The proportion of cracks identified is determined by considering cases where the Dice coefficient for each model exceeds 0.20, indicating a successful crack detection. The improved performance of the reduced weights model suggests that overfitting might have hindered the original model. Consequently, simplifying the model could lead to even higher accuracy in crack detection.

Comparison with the provided Neural Network

Despite the fact that the masks of each image and the output of the reduced weights models does not match up to a great extend, the pimarly goal of this study is to recreate the original model with reduced weights so that it can fit into the fpga bram. So the next comparsion is between the reduced weights models on if they find the cracks so that there can be a winner between the reduced weights models which will be used in the hardware implementation.

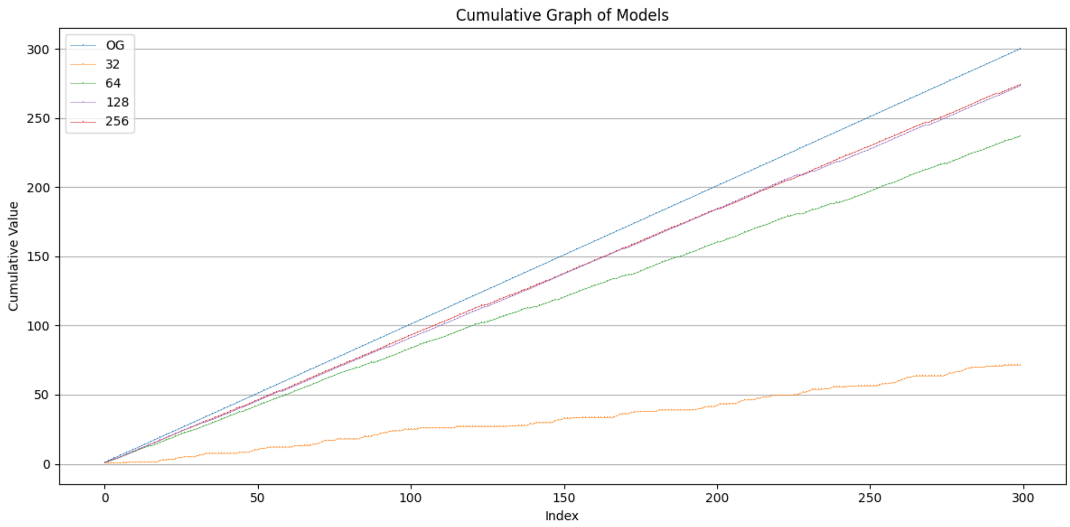


FIGURE 4.9: Overall performance

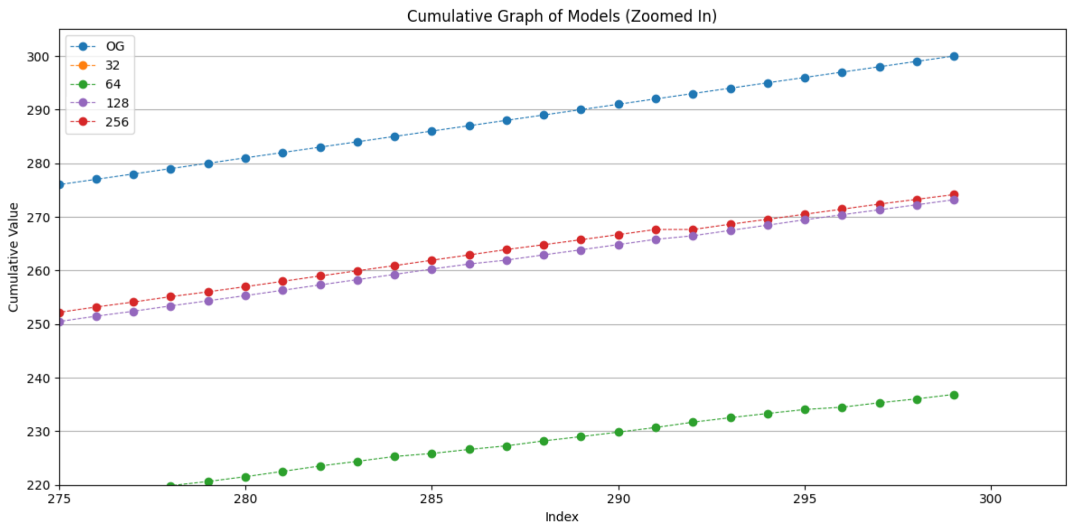


FIGURE 4.10: Zoomed performance

FIGURE 4.11: Diagram of the comparison between overall and zoomed performance

Those Diagrams were calculated as an sum of the dice coefficient of each model, the dice coefficient of the original model were always one because it symbolize perfect functionality, for the others the dice coeeficient value and if no crack was detected both on the original model and the reduced weight model then one was added to both, and if crack was detected by the a reduced weigth model and was not detected by the provided model then 0 was added to this model's performance meter.

Model	Sum of Dice Coefficient	Percentage of similarity	Low	Mean	High
Original	300.00	100.00%	1.00	1.00	1.00
RW 32	71.29	23.76%	0.00	0.09	0.81
RW 64	236.88	78.96%	0.00	0.76	1.00
RW 128	273.19	91.06%	0.00	0.89	1.00
RW 256	274.15	91.38%	0.00	0.90	1.00

TABLE 4.7: Table of the comparison

The table provides a comparative analysis of the four models, employing the Sum of Dice Coefficient and Percentage of Similarity as evaluation metrics. The original model functions as a baseline, exhibiting the maximum attainable values (300.00 and 100.00%, respectively). The remaining models likely signify modifications or transformations applied to the original model. The results demonstrate an anticipated trend: the reduced weight (RW) model with 32 clusters exhibits the lowest degree of similarity to the original model, achieving only 23.76% similarity. As the number of clusters increases, a corresponding increase in similarity is observed. The two models with the highest number of clusters, 128 and 256, demonstrate a marginal difference in accuracy, with the 256-cluster model exhibiting a 91.38% similarity to the original. This suggests a near-identical resemblance between the 256-cluster model and the original. While the 6-bit model offers better storage efficiency, its lower accuracy excludes it from selection for now. However, it could serve as a backup option if storage constraints become a priority.

Next a crack detection test was made where the cracks identified was the same as before, the Dice coefficient for each model exceeds 0.20. As reference point is the original model with 100% crack detection, the resuts are:

Model	Percentage of similarity
Original	100.00%
RW 32	15.62%
RW 64	94.40%
RW 128	96.92%
RW 256	98.83%

TABLE 4.8: Table of percentage of cracks

The anticipated improvement in crack detection accuracy is attributed to a simplified comparison process that focuses on the presence or absence of a crack, rather than its shape. This shift has led to increased accuracy across all models, the two largest clusters models, 256 and 128, exhibit high accuracy, with a particularly notable improvement in the 64-cluster model. The 32-cluster model's performance is noticeably lower than the near-perfect 99% achieved by the 256-cluster model.

Summary

The 256 cluster model has the best accuracy overall, both in crack segmentation and the crack detection test, also the bram occupied when using this model is within the allowed range. So this will be the model which will implemented on the hardware, although in the case of only the detection of the cracks should be the main goal of the project then also the reduced wieght model with the 64 cluster should be considered a good option.

Chapter 5

FPGA Design

5.1 Tools Used

The tools employed in this study are components of the AMD Vitis™ Unified Software Platform, a comprehensive development suite designed for the creation of applications on AMD adaptive SoCs and FPGAs. The platform comprises a comprehensive array of development tools, including the Vivado Design Suite, Vitis HLS (High-Level Synthesis), and the Vitis IDE, all of which played a pivotal role in this study. Furthermore, the platform incorporates additional specialised tools, such as Vitis AI for accelerating machine learning and Vitis Embedded, which is optimised for embedded system development. Collectively, these tools provide a unified and adaptable environment, facilitating the creation of diverse applications across a range of domains.

5.1.1 Vitis High Level Synthesis (HLS)

The AMD Vitis HLS tool[36] enables users to efficiently design complex FPGA algorithms by converting C/C++ functions to RTL. Seamlessly integrated with the Vivado Design Suite and the Vitis unified software platform, Vitis HLS provides a comprehensive solution for heterogeneous system designs and applications. By using directives within the C code, users can tailor the generated RTL to meet specific implementation requirements. The flexibility of Vitis HLS allows different design architectures to be explored from a single C source code, ensuring high quality, correct-by-construction RTL. In addition, C simulation accelerates the design validation process, enabling faster iterations compared to traditional RTL-based simulation. The tool's rich set of analysis and debugging features further facilitates design optimisation.

C-to-RTL Conversion

The Vitis HLS tool effectively transforms C/C++ code into RTL, tailoring its synthesis process to different code components. Top-level function arguments are synthesised into RTL I/O ports, equipped with an interface synthesis hardware protocol for automated implementation. Other C functions are synthesised as RTL blocks, preserving the design hierarchy. To improve performance, C function loops are maintained in rolled or pipelined states. In addition, C code arrays can be strategically mapped to different memory resources, including BRAM, LUTRAM and URAM.

Synthesis reports provide valuable insight into performance metrics such as latency, initiation interval, loop iteration latency and resource utilisation. Vitis HLS pragmas and optimisation directives provide the flexibility to configure synthesis results for the C/C++ code, enabling fine-grained control over the generated RTL.

Simulation and Verification

The Vitis HLS tool includes built-in simulation flows to speed up the verification process. C simulation validates the functionality of the C code, using a C testbench for efficient execution. C/RTL co-simulation reuses the C testbench to ensure that the generated RTL is functionally equivalent to the C source code. The simulation flow integrates analysis, debug and waveform viewing capabilities supported by popular simulator tools.

IP Export

The Vitis HLS tool generates an RTL implementation that can be packaged into a compiled object file (.xo) or exported as an RTL IP. Compiled object files (.xo) are utilized to create hardware acceleration functions within the Vitis application development flow. Alternatively, RTL IP can be employed in other ways like the Vivado IP Integrator Tool or the Vivado IDE.

Pragmas HLS

In addition, the HLS tool offers the option of utilising pragmas [37]to enhance the design, minimise latency, optimise throughput performance, and reduce the area and device resource usage of the resulting RTL code. The aforementioned pragmas can be incorporated directly into the source code for the kernel. The optimisations afforded by the HLS pragmas can be applied on multiple occasions the following section specifies the pragmas employed in this study. The pragmas can be classified into two categories: those used for the purpose of accelerating the process and those used for the purpose of facilitating communication.

Interface

The *pragma HLS interface* is employed to indicate the IP block interface. This is necessary because the extracted IP block will be placed in Vivado, and the outputs and inputs must be known. In total, two of these pragmas are used. Four instances of this specific pragma are observed in the designs.

- *pragma HLS INTERFACE mode = s_axilite bundle = ctrl*

  This line of HLS code defines the CTRL signal as a control signal that will be accessed via the AXI-Lite bus interface

- *pragma HLS INTERFACE ap_ctrl_none*

  The line pragma HLS INTERFACE ap_ctrl_none port=return in the HLS code indicates that the return value of the function will not be employed as a control signal for the synthesised hardware module.

Acceleration

To accelerate the process, Vitis HLS provides specific pragmas that target key aspects of performance optimization. These pragmas help in minimizing the latency, maximizing throughput. By instructing the tool on how to handle loops, memory accesses, and data dependencies, these pragmas enable faster execution of critical sections of the code. The optimization pragmas for acceleration focus on aparallel execution of calculations, which is essential for enhancing the efficiency of hardware designs. Building on the optimization pragmas for accelerating the design, the following were used:

- *pragma HLS pipeline*

  Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. This means that the function or loop can process new data more frequently, improving its overall throughput and efficiency.

- *pragma HLS unroll*

  It can unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

These pragmas were crucial in refining the design for optimal performance and efficient resource utilization.

5.1.2 Vivado

AMD Vivado [38] is a powerful software suite designed for the development and analysis of HDL designs, primarily targeting AMD's FPGAs and SoCs. It provides a unified environment for various stages of the design process, from high-level synthesis to physical implementation. One of the standout features of Vivado is its IP Integrator, which provides a graphical and Tcl-based, correct-by-construction design development flow. This tool supports intelligent auto-connection of key IP interfaces, one-click IP subsystem generation, real-time design rule checks (DRCs), and interface change propagation. Additionally, Vivado offers robust verification and debug capabilities, encompassing design entry, timing analysis, hardware debug, and simulation within a single IDE. Following synthesis and implementation, Vivado generates a suite of reports, including those pertaining to timing, methodology, DRC, utilisation, power, and schematic. These reports are of paramount importance for architects, as they facilitate the validation of design correctness and enable a comprehensive understanding of the underlying implementation details. Once a design has been successfully implemented, a bitstream can be created and exported for programming onto a target platform or utilised within Vitis software for application development.

5.1.3 Vitis IDE

The Vitis Integrated Development Environment (IDE) [39] is distinguished by its highly intuitive and user-friendly graphical user interface (GUI), which has been designed to accommodate a wide range of development workflows. The Vitis IDE offers a consistent and intuitive experience, regardless of whether users are working with traditional project flows or leveraging custom Makefile processes. One of the IDE's most notable attributes is its capacity to transition seamlessly between the graphical user interface (GUI) and the command-line interface (CLI) modes. This affords developers the autonomy to select the environment that optimally aligns with their workflow preferences and project requirements. This flexibility is particularly advantageous for teams and individual developers who may favour different modes of interaction, depending on the complexity and nature of their tasks. The graphical user interface (GUI) mode offers a more visual and interactive approach, which is well suited to those who prefer a straightforward drag-and-drop style interface or are managing complex projects with multiple components. Conversely, the command-line interface (CLI) mode provides a more scriptable and automation-friendly environment, allowing for streamlined project builds, automated tasks, and precise control over compilation and deployment processes. Furthermore, the Vitis Integrated Development Environment (IDE) offers comprehensive integration with a multitude of development tools and libraries, thereby enhancing its versatility across diverse development contexts, including software applications and hardware accelerations. This adaptability renders the Vitis IDE an indispensable instrument for developers striving to innovate across a myriad of platforms and applications, whilst simultaneously maintaining a high level of efficiency and control over their development environment. In essence, the Vitis IDE's dual-mode capability and support for both project and custom Makefile workflows make it an optimal choice for developers who demand a flexible, versatile and efficient development environment.

5.2 Board Used

The AMD Alveo U55C FPGA, used in this project, is a high-performance compute accelerator designed for demanding tasks in high-performance computing (HPC), big data analytics, financial modeling, and machine learning. Built on AMD Adaptive Computing technology, it offers superior processing capability, energy efficiency, and scalability, making it ideal for modern data centers. The Alveo U55C is optimized for compute-intensive tasks with high-bandwidth memory (HBM2) for rapid data access, excelling in industries like finance and scientific research. Its scalability supports integration into HPC clusters, allowing organizations to expand their computing infrastructure without sacrificing performance. Designed for energy efficiency, it reduces operational costs and environmental impact, while its versatility across diverse applications such as climate modeling, signal processing, and big data analytics makes it a valuable tool for addressing complex computational challenges. In short, the Alveo U55C is a powerful, adaptable solution for organizations looking to enhance performance, scalability, and efficiency in their data processing and analytics needs.

FIGURE 5.1: Alveo U55C
https://m.digitalisationworld.com/news/62703/xilinx-launches-alveo-u55c

5.3 Hardware

This chapter provides a detailed outline of the system architecture designed for implementation on the Unmanned Surface Vehicle (USV), specifically focusing on its role in image acquisition and crack detection. A top-down exaplanation is followed, starting with a broad overview of the entire system and then progressively delving into each component. By exploring the architecture from the highest level down to its finer details, the chapter aims to give a holistic and thorough understanding of how the various hardware and software elements—such as the FPGA units and the RISC-V CPU—interact and contribute to the overall functionality of the USV.

5.3.1 Top Level

The system to be integrated into the USV will comprise a Risc-V CPU responsible for managing all vehicle functions. It will also include a camera (not directly involved in the hardware operations) and the hardware system itself, which will execute the CNN upon receiving an image. For the purposes of this study, the key functionalities of the Risc-V CPU will involve capturing images from the camera at a specified resolution, resizing them to 128x128 dimensions, and subsequently writing these resized images to the HBM of the Alveo U55C, which serves as the main memory for the hardware system. Furthermore, the Risc-V CPU will read the output from the hardware system and transmit it to the appropriate destination for crack detection analysis.

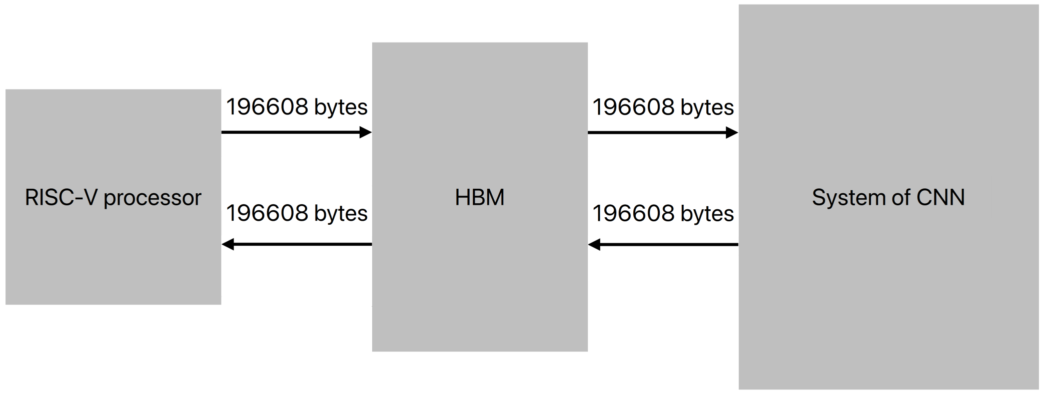


FIGURE 5.2: System of the crack detecting USV

The bytes illustrated above the arrows are calculated like this. After the resizing the dimensions of the image is 128x128. The Risc-V CPU does multiple memory entries, writes 128x128x3 because of the RBG values, each value is a 32 bit float so:

$$(32 \cdot 128 \cdot 128 \cdot 3)/4 = 196608(bytes)$$

Of course there are two arrows because the hardware system return the mask it makes based on the image provided.

The system to be integrated into the USV will comprise a Risc-V CPU responsible for managing all vehicle functions. It will also include a camera (not directly involved in the hardware operations) and the hardware system itself, which will execute the CNN upon receiving an image. For the purposes of this study, the key functionalities of the Risc-V CPU will involve capturing images from the camera at a specified resolution, resizing them to 128x128 dimensions, and subsequently writing these resized images to the HBM of the Alveo U55C, which serves as the main memory for the hardware system. Furthermore, the Risc-V CPU will read the output (the generated mask) from the hardware system and transmit it to the appropriate destination for crack detection analysis.

In the block diagram, two arrows indicate bidirectional communication between the hardware system and the image processing unit. The hardware receives an image and then returns the corresponding mask it generates. The byte values displayed above the arrows represent the size of the data being transferred. Once the image is resized to 128x128 pixels, the Risc-V CPU writes multiple memory blocks, transferring 128x128x3 bytes—accounting for the RGB channels. Since each value is stored as a 32-bit float, the total data transfer can be calculated as follows:

$$(32 \cdot 128 \cdot 128 \cdot 3)/4 = 196608(bytes)$$

5.3.2 Hardware System

The system is all about what this thesis is about. Practically does everything, takes the image form the HBM, provide it to the CNN throught a number of actions and then takes the image generated form the CNN and place it back to the CNN for the Risc-V CPU to have it available.

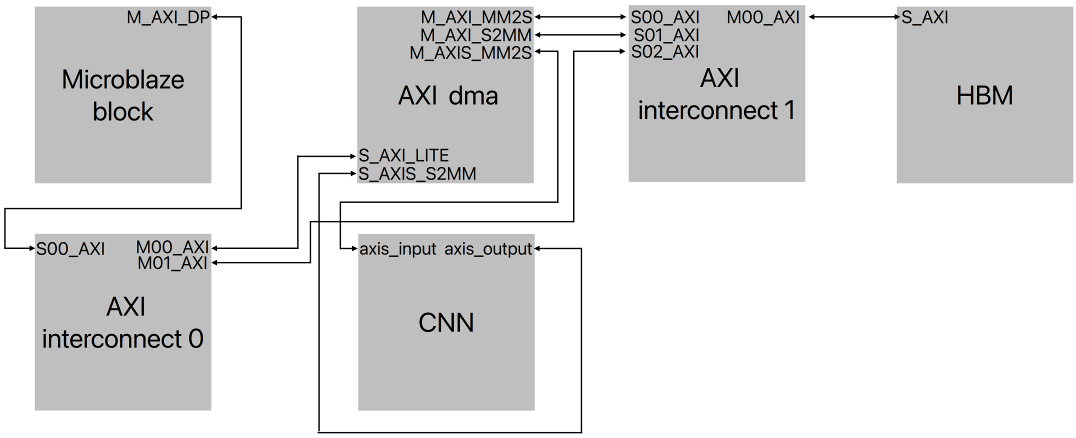


FIGURE 5.3: Hardware System

MicroBlaze

This hardware design utilizes a softcore processor, MicroBlaze, to manage complex communication protocols and memory operations. Softcore processors are a type of processor that is implemented using programmable logic on a hardware platform, such as FPGA, rather than being a fixed part of the hardware. They are described using HDLs, enabling easy modification for specific applications. MicroBlaze's support for various interfaces makes it suitable for complex systems, allowing seamless integration with other programmable logic. While softcore processors may not match the speed of dedicated hardware, their adaptability makes them ideal for scenarios prioritizing design flexibility and rapid prototyping. In this design, MicroBlaze's reconfigurable nature aligns with the system's goals, enabling dynamic reprogramming and component integration.

Axi protocol

The AXI protocol is used to make Microblaze communicate with the other components, the microblaze is the master axi and the memory and DMA are the slave axi. Because there is only one master output available on the microblaze, the block axi interconnect is used, so the Microblaze can communicate with all the devices it must. The axi interconnect is provided some data and an address, based on the address sends the data to the correct block, it acts like a traffic controller, directing data packets between connected devices.

Axi stream protocol

The data transfer protocol employed between the memory and the CNN, as well as from the CNN back to the memory, is the AXI Stream. This protocol is specifically chosen due to its efficiency in handling large image matrices, allowing mathematical calculations to commence as soon as the initial data bits arrive at the CNN module. By facilitating parallel operations, the AXI Stream significantly enhances time efficiency, enabling the CNN to process data in a streamlined and accelerated manner. While the initial architecture does not fully utilize this speed advantage, future iterations of the system will incorporate this optimization to maximize performance and reduce latency.

DMA

The DMA is a feature in computer systems that allows certain hardware devices to access system memory directly, without needing the CPU to manage the data transfer. DMA dramatically speeds up the transfer of large data volumes, especially for devices like network cards and storage drives that frequently handle substantial data chunks. By offloading the intensive task of data transfer to the DMA controller, the CPU is freed to handle other critical operations. This translates to a more responsive system overall. In essence, DMA optimizes system performance by streamlining data transfers and freeing up the CPU, ensuring a smoother and more efficient computing experience.

Hardware system description

The central component of the design is the MicroBlaze system block, which manages the actions of the rest of the hardware system. This block includes several key elements that support the MicroBlaze softcore processor: the microblaze0localmemory, the MicroBlaze Debug Module (mdm1), the clock wizard (clk_wiz), and the reset clock wizard (rst_clk_wiz). Each of these blocks has distinct functionalities crucial to the system's operation. The microblaze0localmemory block serves as local memory for the MicroBlaze processor, allowing rapid access to instructions and data via the processor's data and instruction local memory buses (DLMB and ILMB). A system reset signal (SYSRst) also controls this memory block's operation. The mdm1 module provides debugging capabilities, enabling monitoring and control of the processor during development and troubleshooting. The clk_wiz module generates the necessary clock signals for the system, taking in the CLK input signal and producing an output clock (clkout1) along with a locked signal to indicate stable clock generation. This ensures synchronized operation across various components of the design. The rst_clk_wiz module, on the other hand, manages the reset signals throughout the system. It converts the primary reset signal (RST) into specific reset signals for the processor, buses, and peripherals, ensuring all components start from a known state during initialization or after an error. The design also features two AXI interconnects: axi_interconnect_0 and axi_interconnect_1. The first interconnect (axi_interconnect_0) handles the m_axi signal from the MicroBlaze processor and routes it to the DMA and the second interconnect (axi_interconnect_1). The axi_interconnect_1 interconnect has one M_AXI output that connects to the High Bandwidth Memory (HBM), providing a pathway for data exchanges. It also has three s_axi inputs: two from the DMA and one from the axi_interconnect_0, linking the MicroBlaze interconnect. The axi_dma block acts as the DMA controller, exchanging data with BRAM through an AXI interconnect using two channels: M_AXI_MM2S for receiving data and M_AXI_S2MM for sending data. Additionally, it features an AXI stream port (M_AXIS_MM2S) for sending data to the CNN module. The CNN block receives and sends bursts of data via AXI streams directly connected to the axi_dma, allowing efficient data transfer between the CNN and the HBM. This design showcases a typical FPGA-based embedded system, leveraging a soft processor, memory components, custom accelerators, and peripherals, all interconnected through a robust AXI-based infrastructure.

5.3.3 Convolutional Neural Network

Next the block diagram illustrating CNN architechture is presented.

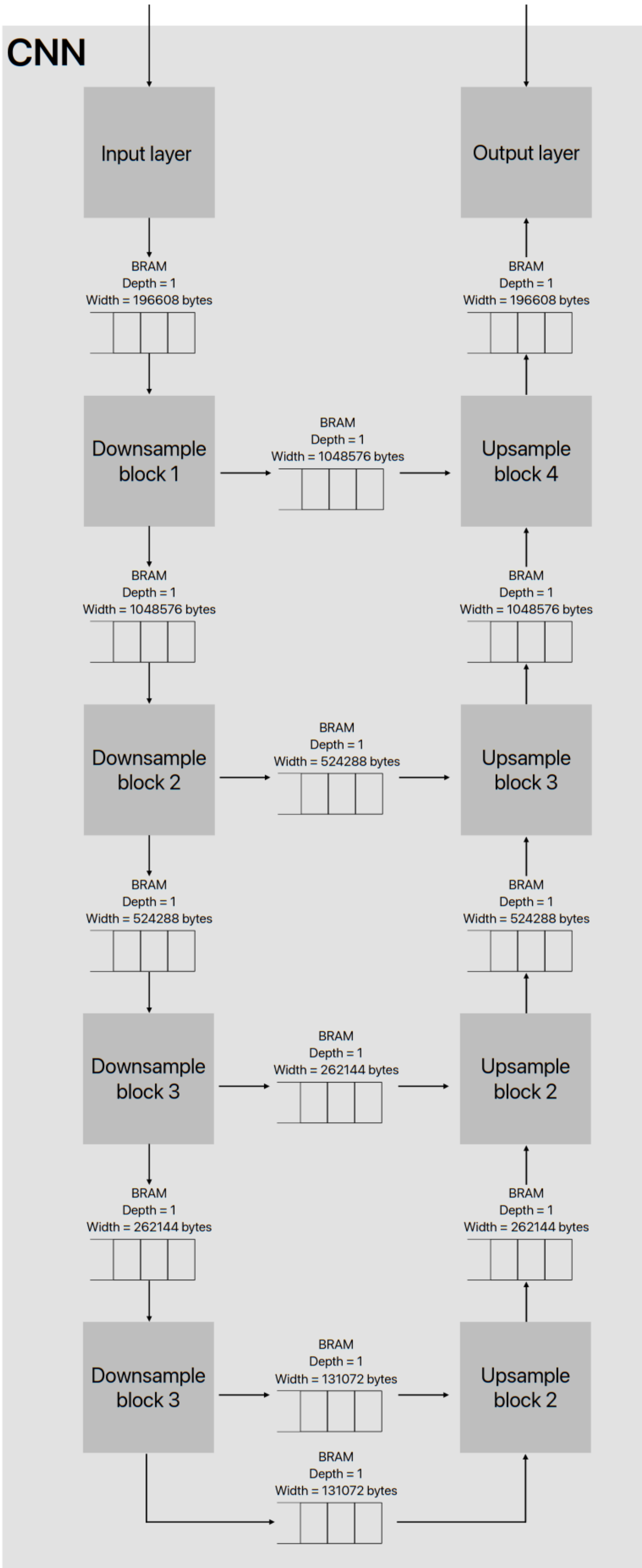


FIGURE 5.4: Block diagram of the CNN

The structure follows the same form described in the previous chapter and indicates the byte size of each layer's output. The input layer's output and the output layer's input both consist of 196,608 bytes, corresponding to a 128x128x3 image, just like the top-level module. Also the first downsample block outputs 1,048,576 bytes, with each subsequent downsample block reducing the output size by half: the second outputs 524,288 bytes, the third 262,144 bytes, and the last 131,072 bytes. Conversely, the upsample layers reverse this process. The first upsample layer outputs 262,144 bytes, the second 524,288 bytes, the third 1,048,576 bytes, and the final upsample layer returns 196,608 bytes. It is noteworthy that the each block communicates with each other via the use of BRAM.

5.3.4 Downsample Block

Below is the block diagram of the first Downsample block. Notably, the inner convolutions within this block produce outputs significantly larger than the final output and input of the Downsample block itself. Diagrams for the remaining three Downsample blocks are not necessary, as they follow the same structure but with different output values. Similar to the blocks in a CNN, the layers within the Downsample block communicate their outputs through the use of BRAM.

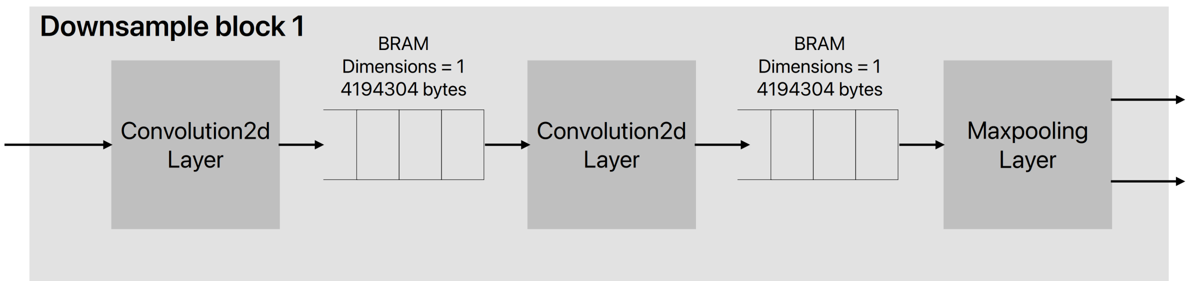


FIGURE 5.5: Block diagram of the CNN

5.3.5 Upsample Block

The final block diagram illustrates the first Upsample block of the CNN. Much like the Downsample block, the layers within the Upsample block transfer their output data through BRAM (Block RAM). Since the structure remains consistent across all Upsample blocks, with only the output values varying, the other three block diagrams are not presented.

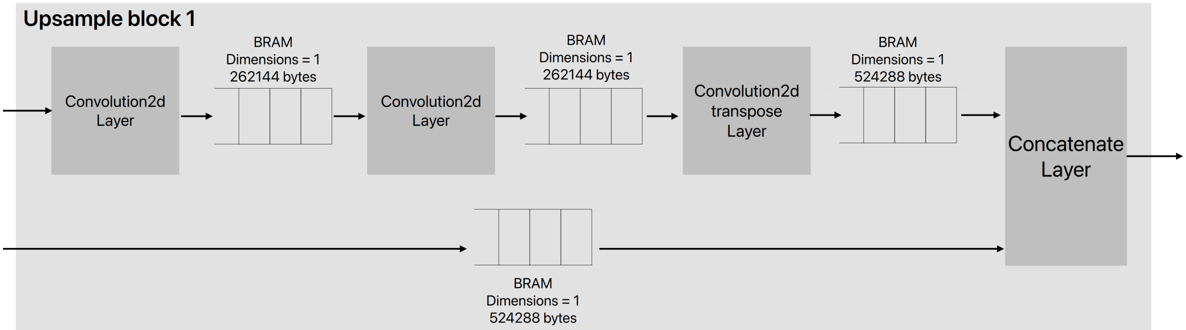


FIGURE 5.6: Block diagram of the CNN

5.4 Methodology

The construction of the provided CNN in HLS necessitates a recompilation of the Python code into C code. The process undertaken was the extraction of the core functions from the Python code, their translation to C, and their placement in the correct order to complete the puzzle. Another significant aspect was the separation of memory for maximum efficiency. This section will therefore break into smaller subsections, each focusing on a significant function of the CNN and the memory separation.

5.4.1 Matrices and Memory fragment

It is not feasible to have a unique set of matrices for each layer and padding, as this would result in a larger data footprint in the BRAM. An optimal approach would be to have a minimum set of matrices that can be reused. Based on this rationale, a total of six matrices are required: the first should be used for the input and output of each convolution, while the second should address the between step of padding. The proposed schematic is as follows:

InOutMatrix(InputOfConv)

↓

Matrix4Pad(PaddingOfInput)

↓

InOutMatrix(OutputOfConv)

Of course the matrices will be one dimensional so that they can fit with every dimensions matrix of every layer. Also the size of these matrices should be specified by the two biggest matrices of the design. The biggest matrix is 64x258x258 = 4260096 elements and the second biggest matrix is 128x130x130 = 2163200, during the padding phases the size of the matrix increases so the biggest value should regard the paddding matrix.

The other four matrices will be about saving the four outputs of the convolution with participation in the concatenation process. The four convolutions hose outputs concatenate are the second the fourth, the sixth and the erighth- which have 1048576,524288,262144 and 131072 elements Correspondingly.

About the bram capacity calculation, the elements of the six matrices should be added.

$$\text{InOutMatrix} + \text{Matrix4Pad} + \text{conv2d1concat} + \text{conv2d3concat}$$

$$+ \text{conv2d5concat} + \text{conv2d7concat} \Rightarrow$$

$$2163200 + 4260096 + 1048576 + 524288 + 262144 + 131072 \Rightarrow$$

$$8389376 \text{ parameters} \Rightarrow$$

$$8389376 \cdot \frac{4}{1048576} = 32.01Mb$$

Adding the megabytes allocated by those matrices to the megabytes allocated by the weights, it is calculated that approximatelly 90 % of bram will be allocated. This leaves 10% to other uses of the fpga or the images given by the risc-v CPU which are 49152 floating point, merely kilobytes. In conclusion, with these matrices every convolution or function can be done within the provided CNN without overloading the bram of the Alveo.

5.4.2 Convolution2d

The algorithm is as previously described in Chapter 4, with the exception that Padded Matrix is now designated Matrix4Pad, and both Ipnut and Output Matrices are now designated InOutMatrix. This modification eliminates the need for a third matrix, thereby reducing storage requirements. The aforementioned convolutions are categorised based on the number of input filters. Seven types were created to replace the 19 convolutions, with the input starting at 3, goes to 64 and then increasing in steps of x2 to 1024(64, 128 etc). The final stage is the convolution_out whose the difference is not only on the input filters.

5.4.3 Convolution2d Transpose

Similarly this function also follows the algorithm discussed in chapter 4 with the smae modifications to the matrices. There are four convolution2d transpose operations, they cant be distinguished based on the number of the input filters, so they are four distinct functions.

5.4.4 Concatenate

The concatenate function undergoes a minor modification, aside from the renaming of the matrices. Four conditional statements are added to ensure the correct matrix is concatenated at each layer. In total, four concatenate functions are executed, each adjusting based on the layer-specific conditions.

Algorithm 9 Concatenate algorithm with cases

```
for Depth do
    for size do
        for size do
            Matrix4Pad[concatIndexA] = InOutMatrix[concatIndexA]
            if layer == 12 then
                Matrix4Pad[k] = conv2d7concat[k]
            else if layer == 16 then
                Matrix4Pad[k] = conv2d5concat[k]
            else if layer == 20 then
                Matrix4Pad[k] = conv2d3concat[k]
            else if layer == 24 then
                Matrix4Pad[k] = conv2d2concat[k]
for size * size * depth * 2 do
    InOutMatrix[ka] = Matrix4Pad[ka]
```

5.4.5 Maxpooling Layer

The construction of maxpooling functions is precisely as described in the preceding chapter.

5.4.6 Image Decoder and Encoder

These two functions are of significant importance as they enable the design to interact with the wider hardware system. In particular, the axi stream protocol exclusively accepts integers as data. Consequently, the image Decoder is responsible for transforming these integers into floating-point numbers, which allows for the execution of mathematical operations. Subsequently, the image Encoder performs the inverse operation, translating the floating-point numbers back into integers for transmission to memory via the axi stream.

5.4.7 Weight Loading

At the initial stage of the CNN, weights are transferred via the AxiStream as compact 32-bit data packets, each containing four 8-bit weight values. The Weight Loading module extracts these 32-bit packets from the stream and unpacks them into four ap_uint<8> weights, which are then stored in the appropriate array.

Chapter 6

Results

This chapter is divided into two sections. The initial section is more technical in nature and discusses the latency of the entire convolutional neural network, as well as that of each individual function, and the resources utilized by the entire CNN. The subsequent section addresses the image quality of the CNN's output.

6.1 Technical Specifications

6.1.1 Timing

The time required for a convolutional neural network (CNN) to process an image was evaluated using synthesis in Vitis High-Level Synthesis (HLS). The synthesis process yields comprehensive data regarding the total number of clock cycles required to complete the entire project, as well as the number of cycles consumed by each individual function within the design. It is anticipated that the convolution operations will constitute the majority of the computational load, consuming over 90% of the total processing time. This is due to the inherently complex and resource-intensive nature of convolutional computations in CNNs. The following table presents the latencies, measured in clock cycles, for each function in the synthesised design. By focusing on these values, the overall efficiency of the implementation can be analysed, with a particular emphasis on optimising the convolutional operations, which are expected to be the primary bottleneck.

Modules & Loops	Latency (cycles)	Latency (Nanoseconds)
LoadWeights	34,513,478	3.45*10^8
ImageDecoder	49,154	4.92*10^5
ImageEncoder	49,154	4.92*10^5
Conv_3	3,145,746	3.14*10^6
Conv_64	86,511,834	8.65*10^8
Matrix_Forward_1	1,048,576	1.049*10^7
Maxpooling	2,120,714	2.121*10^7
Conv_128	289,935,770	2.89*10^9
Matrix_Forward_2	524,290	5.243*10^6
Conv_256	384,307,821	3.843*10^9
Matrix_Forward_3	262,146	2.62*10^6
Conv_512	825,495,572	8.255*10^9
Matrix_Forward_4	131,074	1311*10^6
Conv_1024	1,099,740,169	1.1*10^10
ConvTransp_1024	420,382,728	4.204*10^9
ConvTransp_512	407,405,576	4.074*10^9
ConvTransp_256	99,648,605	9.96*10^8
ConvTransp_128	69,249,404	6.92*10^8
Concatenate	42,270,730	4.23*10^8
Conv_Out	19,402,940	1.94*10^8

TABLE 6.1: Latency statistics for different modules and loops.

The clock period for the design is 10 ns, meaning each clock cycle has a duration of 10 nanoseconds. The total execution time of the project is 9,315,986,106 cycles, which translates to:

$$9,315,986,106 \cdot 10 \cdot 10^{-9} = 93.16 \text{ seconds}$$

This implies that processing each image to determine if a crack is present takes approximately one minute and a half. However, directly summing the cycles of individual functions does not yield this total, as each function is executed multiple times. Each function has an associated coefficient representing how many times it is invoked during the process. To accurately compute the total number of cycles, the following calculations must be performed by incorporating these coefficients into the function latencies derived from the matrix above.

$$
\begin{aligned}
&\text{LoadWeights} + \text{ImageDecoder} + \text{ImageEncoder} + \text{Conv\_3} + 3\cdot\ \text{Conv\_64} \\
&+\text{Matrix\_Forward\_1} + 4\cdot\ \text{Maxpooling} + 4\cdot\ \text{Conv\_128} + \text{Matrix\_Forward\_2} \\
&+4\cdot\ \text{Conv\_256} + \text{Matrix\_Forward\_3} + 4\cdot\ \text{Conv\_512} + \text{Matrix\_Forward\_4} \\
&+2\cdot\ \text{Conv\_1024} + \text{ConvTransp\_1024} + \text{ConvTransp\_512} + \text{ConvTransp\_256} \\
&\qquad +2\cdot\ \text{ConvTransp\_128} + 4\cdot\ \text{Concatenate} + \text{Conv\_Out}
\end{aligned}
$$

This approach allows for an accurate estimation of the total cycles used.

Number meaning

Firstly, some functions are executed multiple times with different input and output matrices. For instance, the maxpooling function is invoked four times, each with progressively smaller matrices. Synthesis reports the latency of the most time-consuming instance among these. This approach is applied consistently across all functions that are called more than once with varying parameters. To simplify computations, the worst-case scenario for each function is considered when calculating the overall execution time.

The weight loader function incurs a significant latency of 34,513,478 cycles, as it loads one weight per cycle. In future iterations of the project, this overhead can be reduced by loading weights only once at the beginning of the process and storing them in block RAM (BRAM) for future access. This optimization would eliminate the need to account for these cycles in the total latency.

It is also observed that convolution functions with larger numbers of weights become increasingly complex due to the associated memory transfers, which result in longer execution times. Interestingly, the initial convolutions, which operate on larger matrices but fewer weights, complete more quickly. This is because, in the current project design, the weights are stored in DRAM rather than BRAM, introducing delays. Consequently, as the number of filters (third dimension) increases, convolution operations take longer. This pattern also applies to the transposed convolutions.

Finally, the encoder and decoder functions, as previously mentioned, convert the result to or from AXI stream format, enabling communication between memory and the processing module. These operations take a number of cycles proportional to the number of pixels in the image.

6.1.2 Resources

In this subsection, the hardware resources utilized in the FPGA implementation are analyzed. Key resources such as the BRAM, DSPs, Flip-Flops, and Look-Up Tables (LUTs) are critical to the performance and efficiency of the design. Understanding the usage of these resources is essential for optimizing the hardware, ensuring scalability, and meeting the timing and area constraints of the FPGA. The following matrix will provide a detailed breakdown of the consumption of each resource type.

Modules	**DSP (%)**	**FF(%)**	**LUT(%)**
Total	26	13	52
Conv_64	∼0	∼0	1
Conv_128	∼0	∼0	2
Conv_256	1	1	4
Conv_512	2	1	5
Conv_1024	16	3	13
ConvTransp_1024	∼0	2	11
ConvTransp_512	∼0	1	6
ConvTransp_256	∼0	1	3
ConvTransp_128	∼0	∼0	2
Conv_Out	∼0	∼0	1
Rest	4	4	3

TABLE 6.2: Resource utilization (DSP, FF, LUT) for different modules.

BRAM

The percentage of BRAM used in this version of the accelerator is only 3% because the weights and input/output matrices of the functions are currently in distributed memory.

LUT

Look-Up Tables (LUTs) are fundamental components in FPGAs, serving as small memory blocks that map specific input combinations to corresponding outputs. Essentially, they define how a given input is translated into an output within the hardware logic. In the context of this design, LUTs are primarily utilized in the convolution functions, including both standard and transposed convolutions. Their usage is particularly concentrated in convolutional layers with a large number of filters. As previously discussed, the greater the number of filters, the more weights are involved in the computation, necessitating additional LUTs to store and process these weights efficiently. The complexity of these layers, driven by the increased memory demands, directly correlates to the higher consumption of LUTs, making them a crucial resource in the implementation of the CNN architecture on the FPGA.

DSP

DSP blocks in FPGAs are specialized hardware components designed to efficiently perform complex mathematical operations, particularly those used in signal processing tasks. They accelerate signal processing tasks, handle multiplications efficiently, offer reconfigurable hardware, optimize resource usage, and support a wide range of applications. These features make DSP blocks in FPGAs a powerful tool for modern digital signal processing needs. The normal convolutions use DSPs, in the initial convolutions the increase in use is linear and on the Convolution_1024 it increases rapidly. DSPs are also not used very often in the transposed convolutions.

FF

Flip-flops are fundamental to implementing pipelining, a technique that allows multiple operations to be processed simultaneously. By using flip-flops at the boundaries of each pipeline stage, you can store intermediate data, enabling different stages of the computation (e.g., convolution, activation, pooling) to operate concurrently. The 13% is not as high as it should, so it indicates that the design does not uses pipeline to the fullest.

6.2 Performance Evaluation

This section presents a comparative analysis of the performance of the hardware accelerator and a software-based implementation of the CNN. To evaluate both approaches, five randomly selected images were processed by the hardware accelerator and the software implementation, which was run using Keras Python libraries. The results from both platforms were analysed to identify similarities and assess the performance of the hardware accelerator. The following images present a detailed discussion of the results, providing a comprehensive examination of any observed differences.

The first image is about the hardware accelerator and the second about the software implementation.

6.2.1 Hardware and Software Results

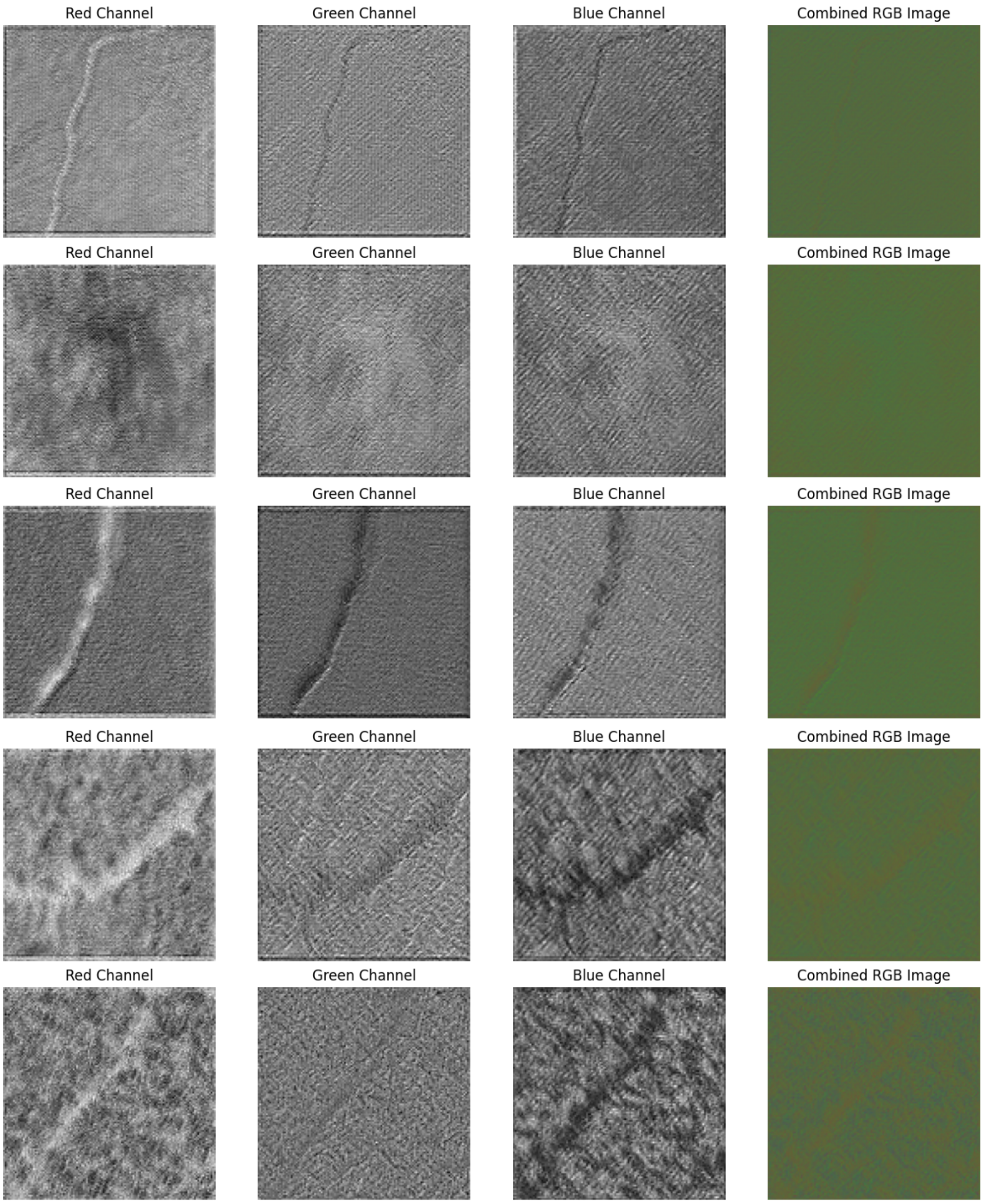


FIGURE 6.1: Hardware Accelerator Resutls

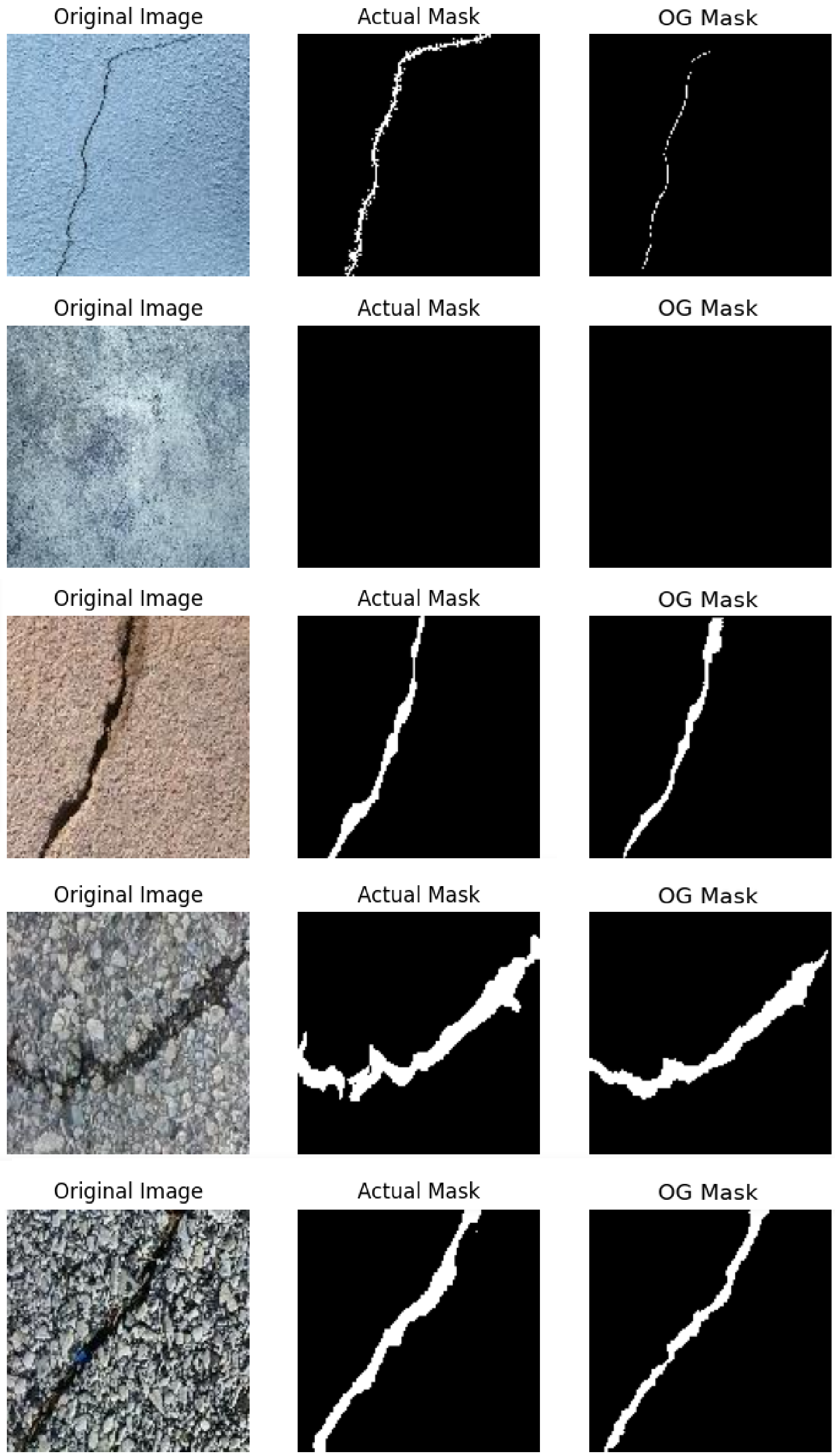


FIGURE 6.2: Software Results

6.2.2 Comparison of Results

In the hardware accelerator results, four images are presented: the first three show the RGB color channels separated, while the fourth combines these channels into a full-color image. Notably, the green channel tends to dominate, overlapping with the other colors. Meanwhile, in the crack segmentation regions, the red channel appears more pronounced, giving it a vibrant red look. Among the channels, the blue channel seems to be the most effective in accurately identifying cracks. In contrast, the software results display three images: the original input image, the corresponding mask provided by the dataset, and the output from the CNN. For better comparison, all five images from the hardware and software outputs are presented in the same order.

Both the hardware accelerator and the software-based CNN effectively detect and segment cracks with a high degree of accuracy. However, a significant difference can be observed: the hardware accelerator's output contains substantial visual noise. In the software-generated results, the image background is entirely black, with only the crack area highlighted in white, offering a clean segmentation. In contrast, while the hardware accelerator makes the crack visible through varying shades of color, its output is affected by considerable noise across all four images. Despite this, the hardware accelerator correctly identified all cracks and successfully ignored the image without any crack, demonstrating its effectiveness despite the noise issue.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This section of conclusions will be seperated into three subsections which will regard, the storage management, the timing and the image quality correspondigly.

7.1.1 Storage Management

In this design, a key strategy for optimizing both performance and memory usage involves the use of k-means clustering alongside the careful application of convolutional operations, which are constrained to just two reusable matrices. A crucial benefit of this approach is that the total memory allocated remains below 64 MB, allowing the matrices to potentially reside entirely within BRAM. This presents an exciting opportunity for significant speedup, as BRAM access is much faster than external memory access.

Although this specific version of the design does not currently use BRAM for matrix storage, the potential for leveraging it remains a pivotal advantage. By doing so, future iterations of the design could exploit BRAM's low-latency, high-bandwidth characteristics to enhance computational efficiency even further.

Moreover, the incorporation of k-means clustering into the design not only minimizes memory consumption but also ensures a high degree of accuracy. The design, when employing k-means, performs exceptionally well and is capable of delivering results comparable to the CNN provided by Almende. This comparison highlights the robustness of the approach, as the use of k-means clustering does not compromise the integrity or accuracy of the convolutional operations.

In summary, while the current implementation doesn't yet capitalize on BRAM storage, the design demonstrates significant promise for optimization. By reducing memory demands and maintaining accuracy through k-means clustering, it provides a strong foundation for future improvements, potentially offering both speed and performance comparable to state-of-the-art CNNs.

7.1.2 Timing and resources

The current timing of the CNN hardware implementation in this version is impractical for real-world applications, such as deployment on an USV. At the current speed, many segments of the port would pass by unnoticed during operation. However, the focus of this thesis is primarily on verifying the functional correctness of the CNN in hardware, rather than optimizing for speed or conducting detailed performance benchmarks.

In terms of speed, one key observation is that the entire CNN takes approximately 93 seconds to complete a single inference. Of this total, the convolution operations—both standard and transposed—account for 74 seconds. This indicates that the vast majority of the execution time is spent on the convolutional layers, which are computationally heavy and central to the network's operation.

Another important factor affecting performance is the low usage of BRAM, which is only utilized at 3%. This underutilization is a major reason for the design's slow speed. BRAM, when effectively utilized, allows for multiple concurrent reads and writes, provided the data matrices are properly partitioned. Maximizing BRAM usage would significantly reduce memory access delays and improve overall performance.

Additionally, none of the other FPGA resources, such as DSPs or LUTs, are utilized beyond 50%, indicating that the design is far from fully exploiting the potential of the hardware. This suggests considerable room for optimization. One area for improvement is the convolution operations themselves, as they are quite similar across the different layers of the CNN. By optimizing the architecture of a single convolution layer, it is likely that the speed of the entire design could be improved.

However, when pursuing such optimizations, it is crucial to ensure that resource utilization remains within bounds, especially when scaling up the convolutional layers. Careful resource management is essential to avoid bottlenecks and ensure that any speed improvements are reflected across the entire design.

7.1.3 Image Quality

First of all When evaluating the image quality produced by the hardware accelerator, two key observations can be made. First, the hardware accelerator is capable of performing crack segmentation to a considerable extent. However, due to the significant amount of noise present in the output, it is difficult to accurately assess its performance using traditional metrics such as the IoU or Dice coefficient. This noise complicates the evaluation of segmentation accuracy.

A potential solution to this issue could involve incorporating a denoising algorithm after the hardware accelerator's output. If the overall performance and timing of the system improve significantly, this denoising step could be handled by a RISC-V processor, helping to refine the image quality before final analysis. In terms of real-world application, the USV does not require extremely high frame rates. For instance, achieving 60 FPS is unnecessary. A frame rate of 1 FPS, or even lower, would be more suitable and provide sufficient time for crack detection during operation without overwhelming the system.

Another important observation relates to the color channels used in the output image. Not all color channels are necessary for detecting cracks. For example, the red or blue channels alone are effective in identifying cracks, while the green channel and the combined color image make it more difficult to discern crack patterns. This suggests that simplifying the output to focus on the more informative color channels could streamline the detection process, potentially reducing computational complexity and improving accuracy.

7.2 Future Work

- **Make it faster, with the use of BRAM and piepline:** A primary method to enhance FPGA performance is optimizing memory usage and computation timing. One way to achieve this is by loading the weights of the neural network into the FPGA's BRAM. By storing the weights in BRAM, it reduces the reliance on slower external memory, minimizing latency. Moreover, implementing pipelining techniques allows different stages of computation to overlap, so the hardware can process multiple operations simultaneously, reducing the overall time per operation. This would enhance both throughput and execution speed significantly.
- **Separate convolution and weight matrices:** In the existing implementation, it may be advantageous to examine the distinct handling of convolution operations and weight matrices. Rather than utilising a single function multiple times, separating the convolution operations from the weight matrices permits the independent optimisation of each. The rationale is that the matrix of weights associated with each convolution can be broken into smaller matrices in a manner that facilitates the parallelisation of tasks. This can enhance the flexibility of the system by reducing data dependencies and enabling optimisations.
- **Bitstream: Run it in Alveo:** An important step for scaling the implementation would be generating a bitstream for the design and deploying it on a high-performance FPGA accelerator, such as Xilinx Alveo. Alveo cards are designed for AI workloads and offer substantial parallelism, which would be beneficial in handling complex computations in real-time. Running the bitstream on Alveo would allow for real-world testing, enabling the system to process larger datasets with higher throughput and validate its performance in practical applications.
- **Inspect the Vitis HLS directives further:**The Vitis High-Level Synthesis (HLS) tool offers various optimization directives (pragmas) that can significantly impact the performance of FPGA designs. A deeper exploration of these directives such as array partitioning, bind storage, or dataflow optimization, could help further optimize resource utilization and timing. Applying these pragmas judiciously can streamline operations, reduce bottlenecks, and ensure that the hardware is fully utilized, improving overall latency and efficiency.

- **Reuse resources of the design** In FPGA designs, resource reuse between layers is essential when the design targets latency rather than throughput. Instead of assigning separate resources (such as DSP slices or logic elements) to each layer, the same hardware can be reconfigured and reused across different layers of the neural network. This approach maximizes the use of available FPGA resources, reducing the total resource footprint and enabling more complex models to run on the hardware without exceeding capacity, while maintaining low latency.

References

[1] *Machine Learning Explained*. URL: https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/.

[2] *Neural Networks Explained*. URL: https://www.ibm.com/topics/neural-networks.

[3] *Convolution Neural Networks Explained*. URL: https://www.arm.com/glossary/convolutional-neural-network.

[4] *Types of Convolution Layers Explained*. URL: https://www.databricks.com/glossary/convolutional-layer.

[5] *Types of Convolution Layers Explained*. URL: https://iq.opengenus.org/linear-activation-function/.

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

[7] *Fully Connected Layers Explained*. URL: https://eitca.org/artificial-intelligence/eitc-ai-dlptfk-deep-learning-with-python-tensorflow-and-keras/convolutional-neural-networks-cnn/introduction-to-convolutional-neural-networks-cnn/examination-review-introduction-to-convolutional-neural-networks-cnn/what-is-the-role-of-the-fully-connected-layer-in-a-cnn.

[8] *UNet model Explained*. URL: https://www.analyticsvidhya.com/blog/2022/10/image-segmentation-with-u-net.

[9] *Residual Neural Network Explained*. URL: https://en.wikipedia.org/wiki/Residual_neural_network.

[10] *Vgg Neural Network Explained*. URL: https://deepchecks.com/glossary/vggnet.

[12] *Uniform and Non Uniform Quantization Explained*. URL: https://anamma.com.br/en/uniform-vs-nonuniform-quantization/.

[13] *K-means Explained*. URL: https://en.wikipedia.org/wiki/K-means_clustering.

[14] *Logarithmic Quantization Explained*. URL: https://collab.dvb.bayern/download/attachments/75112203/fp_berger.pdf?version=1&modificationDate=1644482642003&api=v2.

[15] *Vector Quantization Explained*. URL: https://speechprocessingbook.aalto.fi/Modelling/Vector_quantization_VQ.html.

[16] *Companding Quantization Explained*. URL: https://electronicscoach.com/companding.html.

[19] Kiran Seshadri et al. *An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks*. 2022. arXiv: 2102.10423 [cs.LG]. URL: https://arxiv.org/abs/2102.10423.

[20] Ahmad Shawahna, Sadiq Sait, and Aiman El-Maleh. "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review". In: *IEEE Access* PP (Dec. 2018), pp. 1–1. DOI: 10.1109/ACCESS.2018.2890150.

[23] Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. "Performance and Scalability of GPU-Based Convolutional Neural Networks". In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 2010, pp. 317–324. DOI: 10.1109/PDP.2010.43.

[24] Gousia Habib and Shaima Qureshi. "Optimization and acceleration of convolutional neural networks: A survey". In: *Journal of King Saud University - Computer and Information Sciences* 34.7 (2022), pp. 4244–4268. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2020.10.004. URL: https://www.sciencedirect.com/science/article/pii/S1319157820304845.

[25] Linus Pettersson. *Convolutional Neural Networks on FPGA and GPU on the Edge: A Comparison*. 2020.

[26] Yixing Li et al. *A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks*. 2017. arXiv: 1702.06392 [cs.DC]. URL: https://arxiv.org/abs/1702.06392.

[27] Walther Carballo-Hernández, Maxime Pelcat, and François Berry. *Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks?* 2021. arXiv: 2102.01343 [cs.AR]. URL: https://arxiv.org/abs/2102.01343.

[28] Nicolò Ghielmetti et al. *Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml*. 2022. arXiv: 2205.07690 [cs.CV]. URL: https://arxiv.org/abs/2205.07690.

[29] Tim Chisholm, Romulo Lins, and Sidney Givigi. "FPGA-Based Design for Real-Time Crack Detection Based on Particle Filter". In: *IEEE Transactions on Industrial Informatics* 16.9 (2020), pp. 5703–5711. DOI: 10.1109/TII.2019.2950255.

[30] Stephen L. H. Lau et al. "Automated Pavement Crack Segmentation Using U-Net-Based Convolutional Neural Network". In: *IEEE Access* 8 (2020), pp. 114892–114899. DOI: 10.1109/ACCESS.2020.3003638.

[31] Alessandro Di Benedetto, Margherita Fiani, and Lucas Matias Gujski. "U-Net-Based CNN Architecture for Road Crack Segmentation". In: *Infrastructures* 8.5 (2023). ISSN: 2412-3811. DOI: 10.3390/infrastructures8050090. URL: https://www.mdpi.com/2412-3811/8/5/90.

[32] Fangzheng Lin et al. "Crack Semantic Segmentation using the U-Net with Full Attention Strategy". In: *CoRR* abs/2104.14586 (2021). arXiv: 2104.14586. URL: https://arxiv.org/abs/2104.14586.

[33] Rishabh Goyal et al. *Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms*. 2021. arXiv: 2102.02147 [cs.CV]. URL: https://arxiv.org/abs/2102.02147.

[34] Sean I. Young et al. "Transform Quantization for CNN Compression". In: *CoRR* abs/2009.01174 (2020). arXiv: 2009.01174. URL: https://arxiv.org/abs/2009.01174.

[36] *Vitis HLS Explained*. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[37] *Pragmas HLS Explained*. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas.

[38] *AMD Vivado Explained*. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.

[39] *Vitis IDE Explained*. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-ide.html.