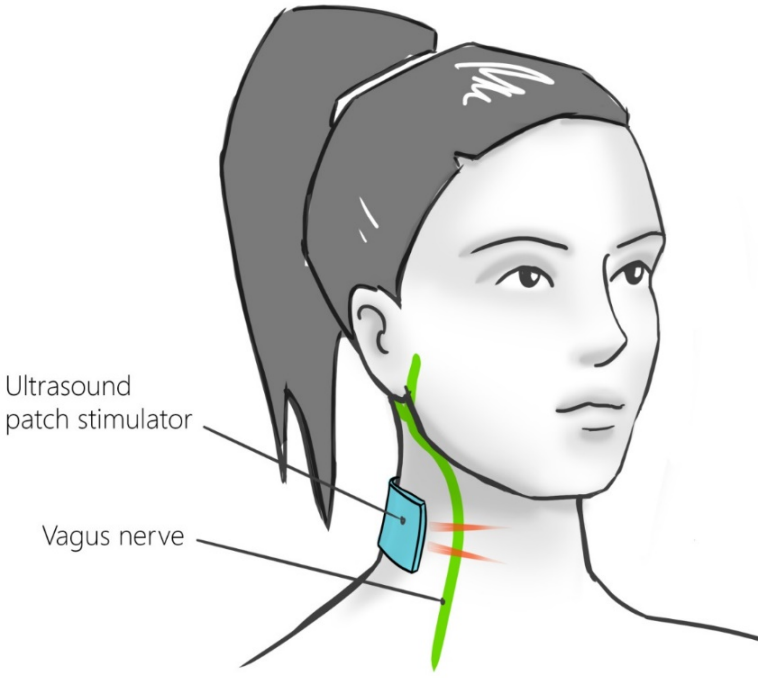
OBJECT DETECTION INSIDE A WEARABLE ULTRASOUND NEUROMODULATOR PATCH

A DEEPER LOOK INTO IMPLEMENTING ULTRASOUND NEUROMODULATOR PATCHES AND HOW TO FIND NERVES USING COMPUTER VISION TECHNIQUES



OBJECT DETECTION INSIDE A WEARABLE ULTRASOUND NEUROMODULATOR PATCH

A DEEPER LOOK INTO IMPLEMENTING ULTRASOUND NEUROMODULATOR PATCHES AND HOW TO FIND NERVES USING COMPUTER VISION TECHNIQUES

By

Christiaan Boerkamp

in partial fulfilment of the requirements for the degree of

Master of Science
in Biomedical Engineering
Specialization: Medical Devices and Bioelectronics

at the Delft University of Technology,
to be defended publicly on Wednesday January 26, 2021 at 1:00 PM.

Project duration:	October 1, 2020 – January 26, 2022	
Supervisor:	Dr. Tiago Costa,	TU Delft
Thesis committee:	Prof. Dr. Wouter Serdijn,	TU Delft
	Dr. Tiago Costa,	TU Delft
	Dr. Borbála Hunyadi,	TU Delft

This thesis is confidential and cannot be made public until October, 2023
An electronic version of this thesis is available at http://repository.tudelft.nl/.

Science is a wonderful thing
if one does not have to earn one's living at it.

Albert Einstein

CONTENTS

List of Figures

List of Tables

List of Abbreviations

1D	One-Dimensional
2D	Two-Dimensional
ANN	Artificial neural network
ASIC	Application-Specific Integrated Circuits
AU	Arithmetic Unit (synonym of Processing Engine)
AXI	Advanced Extensible Interface
BNN	Biological Neural-Networks
BRAM	Block Random-Access Memory
CNN	Convolutional Neural Networks
CMA	Continuous Memory Allocator
CPU	Central Processing Unit
CU	Compute Unit
DF	Dataflow
DMA	Direct Memory Access
DNN	Convolutional Neural Networks
DRAM	Dynamic Random Access Memory
FC	Fully Connected
FF	Flip-Flops
FIFO	First In, First Out
FP	Floating Point
FPS	Frames Per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis Language
im2col	Image to Column (algorithm)
Im2Col	Image to Column (operator)

INT	Integer
IO	Input/Output
IP	Intellectual Property
LKF	Linear Kaiman-Filter
LUT	Look-Up Table
LUTRAM	LUT (Distributed) Random-Access Memory
MAD	Mean Absolute Difference
MPSOC	Multiprocessor Systems-On-Chips
MSE	Mean Squared Error
MVAU	Matrix Vector Activate Unit
NCC	Normalized Cross-Correlation
NN	Neural Network
PE	Processing Engine
RAM	Random-Access Memory
ReLU	Rectified Linear Unit
RTL	Register-Transfer Level
SDWC	StreamingDataWidthConverter
SIMD	Single Instruction, Multiple Data
SSE	Sum of Square Error
SWU	Sliding Window Unit
VLV	Vivo Las Vagus
VVAU	Vector Vector Activate Unit, Vector_Vector_Activate_Batch
YOLO	You Only Look Once

SUMMARY

Neuromodulation of the vagus nerve is used as a treatment for all kinds of ailments and even as a means of improving the wearer's physiology, however, this form of treatment is not popular due to its invasive nature, high chance of side effects, and short period between reimplementation surgery, as such an alternative is sought in the form of neuromodulation using ultrasound. In this thesis, 2 designs for wearable ultrasound neuromodulators for the vagus nerve are suggested, based on these designs a solution is made, solving the problem of detecting vagus nerve within a wearable environment. To detect the vagus nerve two methods are proposed: neural networks and template matching. Based on these methods and these proposed designs 5 unique works have been created, Vivo las vagus (VLV) an object detector using a neural network, a mobile implementation of both VLV as well as template matching, an FPGA implementation of template matching, and 2 FPGA implementations of VLV in which one uses a streaming dataflow architecture and the other a systolic array architecture.

The best results are achieved with the streaming dataflow architecture implementation of VLV within an FPGA, resulting in an accuracy of 87.5 percent on the test set with the 10.88 FPS/watt, and inference of 0.174 seconds. This was achieved by using FINN, a community project for converting software neural networks into HDL representation for the FPGA. Combined with to the best of the author's knowledge, first-ever created loss function to automatically decrease the bit width of a quantized neural network layer without impacting the accuracy during training creating the first-ever fully automated end to end flow for creating a software neural network object detector and converting it towards an HDL representation, allowing biomedical engineers without knowledge of digital electronics or Neural networks to simply load in data and run the python files. To assess the accuracy an accuracy calculation function was created together with a dataset and test set with images taken from [1]. As the dataset has shown to be lacking severely in variety the accuracy assessment of all of the implementations can be considered moot. The VLV FINN implementation was compared to other FPGA implementations based on energy efficiency, showing that the work created within this thesis is one of the best in terms of power efficiency and the smallest in terms of resource usage footprint.

ACKNOWLEDGEMENTS

Let me start by saying that these past two years have been hard on everyone, despite the almost normalization of the restrictions and lockdowns it can still cause distress, especially for students in an isolated environment. This is why I would like to start with thanking my friends, and family for always supporting and helping me, even if it is via a small gesture such as a nice message during the holidays or the occasional coffee, the positive effects it had on me cannot be understated.

I would like to thank my parents and my sister especially as they always look out for me and support me, and despite a long travel time always show up when I needed them, even if it was me calling up in a panic asking for help to fix the water pipes in my apartment.

It is no secret that I as a biomedical engineer chose a subject far outside of my traditional field with the focus on digital electronics, neural networks, and computer vision, which has taught me that is more than OK to swallow your pride and ask other people who have more expertise then you for their opinion. I have met many acquaintances and I would even go as far as to call some of them my friends, and I would recommend anyone reading this in the position of working on their master thesis to not be afraid to ask for help.

As someone who is not from an academic background the writing of a thesis was not always the easiest subject for me, as such I would like to thank Jia Jun Yeh, Medha Krishnaswamy, and Akhil John Thomas for taking time out of their schedule and helping me with feedback and suggestions and being general good guys.

I would also like to thank both the NNgen team and the FINN team for their amazing contributions to their open source community projects. These projects truly help with making niche subjects such as neural networks more accessible for a wider audience and such efforts can only be applauded.

The TU Delft and specifically the bioelectronics group have always been supportive of my ideas and as such I would to thank the team headed by prof.dr.ir. Wouter A. Serdijn for their help both mental, fiscal, as well as just being general good guys to be around. A special thanks to Rohan Brash and Samprajani Rout for keeping me company during the long days and nights in the lab.

Finally and most importantly I would like to thank my Supervisor Dr. Tiago Costa, who was with me during the entire thesis project from beginning to the end, helping me with all facets included within the thesis, be it helping me with writing a thesis, dealing with

my general moments of incompetence, giving his expert opinion digital electronics, or simply teaching me how to present for a wide audience Tiago's help proved invaluable. The entire idea of a wearable ultrasound patch was his idea and he gave me all the room in the world for me to realise my vision, as such this entire thesis would simply not have been possible without him. Finally, I would like to thank him for always reminding me to use passive voice in my thesis except for the acknowledgements in which I cite this as one of his examples: "I am honored to have worked with such a gifted supervisor".

As such I would close my acknowledgements with these final remarks.

I am honored to have worked with such a gifted supervisor, I will buy you a beer after my defence.

Christiaan Boerkamp
Delft, January 2022

1

INTRODUCTION

1.1. INTRODUCTION

THIS thesis will cover the implementation of object detection of the vagus nerve within the environment of a wearable ultrasound neuromodulator, the neuromodulator in question will be used to stimulate the left vagus nerve. A neuromodulator artificially stimulates a neuron in order to induce an action potential, this artificial stimulation is referred to as neuromodulation. Neuromodulation of the vagus nerve is used as a treatment for all kinds of ailments such as depression [17], Rheumatoid Arthritis [18], and even as a means of improving the wearer's physiology [19–21], however, this form of treatment is not popular due to its invasive nature, high chance of side effects, and short period between reimplementation surgery [22–25], as such an alternative is sought in the form of neuromodulation using ultrasound, as in theory this would negate all these negatives brought caused by traditional lead based neurmodulation. The following criteria must be met for the neuromodulator to be considered wearable:

1. The neuromodulator needs to be able to stimulate the nerve chosen by the wielder.

2. The neuromodulator needs to be able to make an ultrasound image of the nerve and surrounding tissue.

3. The neuromodulator should be able to freely sweep the target area by forming a beam of ultrasound using a phased ultrasound transducer array.

4. The neuromodulator should be able to perform its tasks autonomously.

5. The user should not experience any discomfort from wearing the neuromodulator or be perceived as cumbersome.

6. The neuromodulator should be able to find the nerve with high accuracy and within a small time frame.

1

As the criteria are too large and varied for a single masters thesis a focus is laid on finding the vagus nerve inside of a wearable ultrasound neuromodulator. After all, a neuromodulator cant stimulate a nerve if it does not where it is positioned. As there are not yet fully wearable ultrasound neuromodulators 2 hypothetical designs are proposed with 2 different methods of implementing wearable object detection of the vagus are proposed within the literature review of this thesis, one using an FPGA and one using an Android phone, both using computer vision techniques to identify the vagus nerve inside a B-mode ultrasound image. Based on these hypothetical designs unique implementations for wearable object detection of the vagus nerve are implemented.

1.1.1. OVERVIEW VAGUS NERVE

The vagus nerve is known as the 10^{th} cranial nerve ([26], [27]) and stretches throughout the body connecting to organs and muscles, making it the most extensively connected distributed cranial nerve in the body. Two main branches can be found for the vagus nerve, the left vagus nerve, and the right vagus nerve. Both branches run symmetrically to one another from the carotid sheath down to the chest after which they branch down asymmetrically down the gut. As seen in Figure 1.1 the vagus nerve runs parallel to the common carotid artery (CCA) and the Jugular Vein (JV).

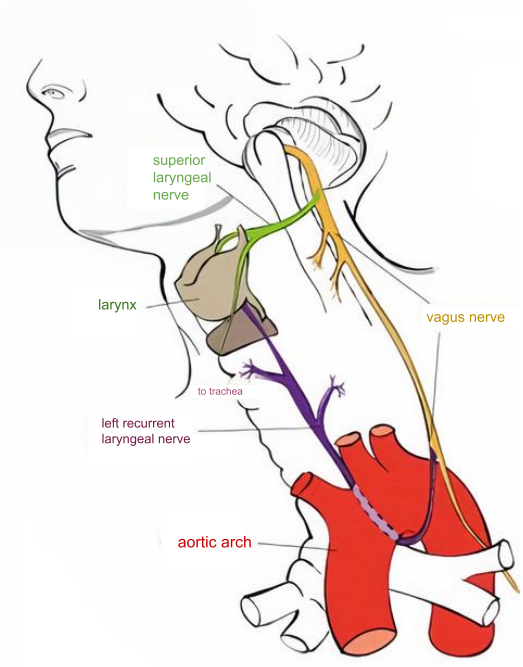


Figure 1.1: A drawing depicting some of the organs present in the neck.

The Vagus nerve physiological functionality can be summed up with three main objectives: Providing signals to skeletal muscles, controlling the parasympathetic nervous system, and acting as an information highway between the brain and the enteric nervous

system. The amount of skeletal muscles controlled by the vagus nerve is quite limited, mainly the soft pallet of the tongue and the Larynx.
The sympathetic nervous system is mainly controlled by the nerves of the spinal cord providing the 'fight or flight ' response, in contrast, the parasympathetic nervous system provides the 'rest and digest 'response of your body making it the antagonist of the sympathetic nervous system. The rest and digest response translate itself functionally into an increase of blood vessel dilation around the organs of the digestive tract allowing for easier uptake of nutrients, a decrease in heart rate, and a constriction of the blood vessels surrounding many skeletal muscles. Sexuality arousal in both men and women is the last major component that is regulated via the parasympathetic nervous system as it controls muscles within the genitals needed for erection, lubrication, and ejaculation.
The last major physiological functionality provided by the vagus nerve is its usage as a connection between the brain and the enteric nervous system giving it the synonym of the brain-gut axis. The enteric nervous system can be seen as a huge amount of receptor- and effector-cells working independently from the brain or the spinal cord. Many of these receptors are lined across the gastrointestinal tract. These receptor cells can for example influence the control of food intake and regulation of satiety, gastric emptying, and energy balance by transmitting information toward the solitary tract in the brain. Another example of receptor functionality is how intestinal receptor is influenced by microbiota, these influences as recent studies have shown might cause anxiety and depressive-like behaviors.

1.1.2. STIMULATION OF THE VAGUS NERVE

Stimulation of the vagus nerve is a medical procedure that has been practiced in western medicine for thousands of years. First mentioned by Hippocrates it is suggested that stimulating the Vagus nerve through massaging has relaxing effects and causes deep sleep in the patient. ([28])
In the 21st-century, vagus nerve stimulation has become more elegant not requiring anymore rough massaging, instead, the usage of an electro stimulator implant has become the defacto neuromodulation method. First introduced in 1988 the implant consists of a waveform generator sending electrical impulses down a lead wire which ends up in leads coiled around the left vagus nerve, this entire device is implanted under the skin of the patient. The reason that the left vagus nerve branch is chosen compared to the right branch is that the electric stimulation in the right branch can cause unwanted heart rate decreases as a side effect. [29]
The surgery required to surgically implant this device is therefore classified as a 'risky surgery' [22] reducing the eligible patients by a large factor. For those who are eligible for the implementation surgery around 70% percent experience side-effects from the implant ranging from Voice alteration to infection [23], [24]. This problem combined with a high re-implementation rate of 47 percent causes the usage of a vagus nerve neuromodulator implant to be seen as a last resort therapy [25].

For all its drawbacks vagus nerve stimulation has shown to alleviate symptoms of- and even combat diseases and disorders plaguing the patient. For example, vagus nerve implementation has been shown to combat the following diseases:

1

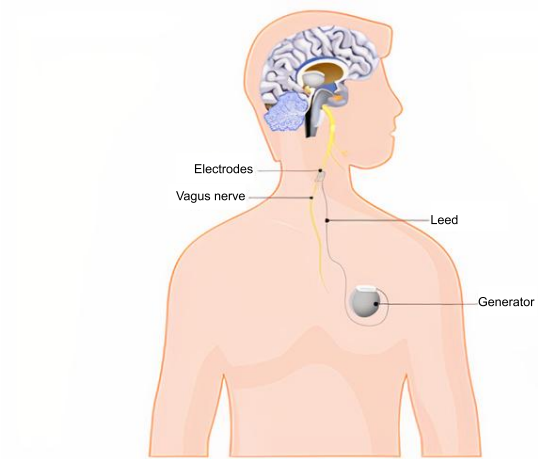


Figure 1.2: Drawing depicting typical implementation of traditional lead based vagus nerve stimulator.

- Vagus nerve stimulation is now an accepted treatment for epilepsy, and it is shown that after a period of 18 to 24 months period the improvements are made in the quality of life for patients suffering from epilepsy. [17]
- There are promising results that show how vagus nerve stimulation over 12 months decreases the severity of depression [30]
- Vagus nerve stimulation in patients suffering from multiple scoliosis has been shown to improve dysphagia and head postural cerebellar tremors resulting in improvements in swallowing of fluids and 'piecemeal' deglutition. [31]
- Vagus nerve stimulation at patients suffering from Chrohn's disease over 6 months has been shown that in a significant percentage of test subjects there are major improvements in the quality of life. [32]
- Rheumatoid Arthritis and perhaps even more autoimmune and autoinflammatory diseases have a decrease in severity due to the vagus nerve stimulation causing a decrease in tumor necrosis factor production. [18]

Vagus nerve stimulation is also shown to produce improvements in the physiology of an unafflicted human being. For example, the following improvements in the physiology of test subjects have been recorded.

- It is shown that vagus nerve stimulation enhances recognition memory. ([19])
- Vagus nerve stimulation is shown to boost the drive to work for rewards. ([20])
- The immune system response has been shown to improve under vagus nerve stimulation, which could with elevation against diseases such as COVID 19 ([21])

1.1.3. ALTERNATIVES TO IMPLANTS

Vagus nerve implants, while producing promising results in combatting diseases and improving human physiology cause severe side-effects which limit the number of users to a large degree. For positives of stimulation to be used without any of the negatives associated with the implementation of the implants an alternative procedure for stimulation needs to be used.

One of these alternatives is ultrasound neuromodulation which provides the following improvements compared to traditional stimulation using implants as the stimulation medium: [1, 3, 33]

- Non-invasive: the implementation of an ultrasound neuromodulator does not require any surgery and can be easily applied and removed from the human body.
- Easily replaceable: Due to the non-invasive nature of an ultrasound neuromodulator device, the ability to replace a device (due to breaking for example) can be done easily.
- No electrode-based side-effects: Thanks to the usage of ultrasound there is no need to connect electrodes to the nerves, ergo there are no electrode-based side-effects such as infection.
- High depth of penetration: Modern ultrasound transducers can reach high depths in soft human tissue allowing them to reach nerves deeper inside the human body.
- High precision: Ultrasound transducer can reach high resolutions up to micrometers allowing stimulation of specific nerves and negating unwanted activation of surrounding nerve tissue.

As the usage of ultrasound negates the negatives from traditional implants while having its unique positives the question that arises is “How is a wearable ultrasound vagus nerve stimulator implemented?”. To answer this question a literature review has been made which is detailed in the following pages.

2

LITERATURE

2.1. INTRODUCTION

WITHIN this chapter a literature review is given surrounding the implementation of a wearable ultrasound neuromodulator. To understand the nature of ultrasound neuromodulators a small section has been dedicated surrounding the ultrasound mechanics of imaging and neuromodulation. A review has been performed on the works of [1], [2], [3], in which within this literature review criteria are established and used to judge the merits of every work. This review is used as a basis for the creation of two proposed designs taking inspiration from the previous works.

Due to the experimental nature of this research topic, literature has been mostly provided by the mentor of the author of this thesis. More conventional background information is gathered from Google scholar and books.

The following terms were used within the literature search of Google Scholar.

- Depth Vagus nerve (1 useful)
- Vagus nerve physiology(3 useful)
- Vagus nerve Anatomy(2 useful)
- Ultrasound physics(2 useful)
- Vagus nerve Side-effects(3 useful)
- Vagus nerve Stimulation (7 useful)
- Ultrasound Nerve interactions(3 useful)
- Wearable ultrasound Neuromodulator(4 useful)

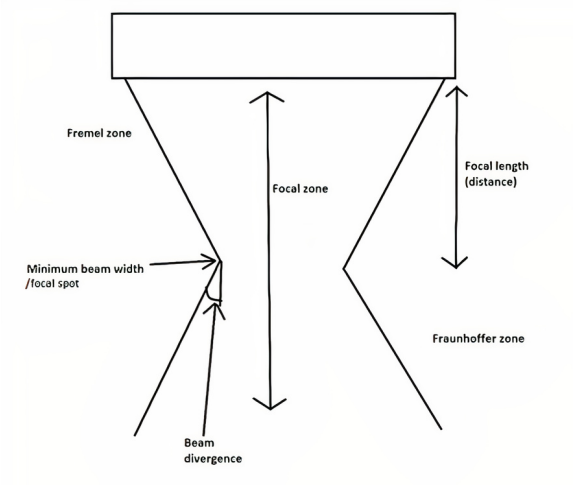


Figure 2.1: Figure depicting an ultrasound beam with all separate components annotated.

2.2. ULTRASOUND MECHANICS

Both imaging and neuromodulation basic principles are based on the forming of an ultrasound beam [34]. An ultrasound beam consists of a Fremel zone, a Focal spot, and a Fraunhofferzone (see Figure 2.1). There are multiple ways to form an ultrasound beam, one of which is to use a phased array.

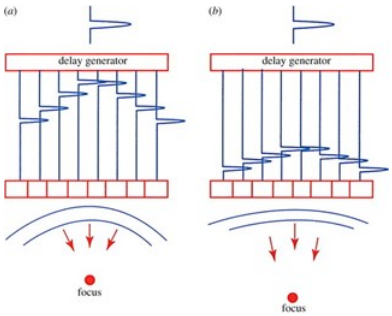


Figure 2.2: Figure depicting the influence of delay generator in the forming of an ultrasound beam and the position of the focus spot.

A phased array is a device that consists of multiple smaller transducer elements, these transducers transmit timed acoustic waves that constructively interfere due to the Huygens–Fresnel principle forming an acoustic ultrasound beam. [34]
The Huygens–Fresnel principle describes how a wave can consist of multiple smaller waves called wavelets. These wavelets can positively or negatively interfere with each other causing a shaped larger wave to form. In Figure 2.2 there is an example shown, in which phased array A has larger delays between the wavelets causes the larger wave

to be shaped as a convex with a deep curve causing the focal spot to be close to the transducer. In comparison phased array B has short delays between the wavelets causing the resulting larger wave to have a more shallow convex shape with a focal spot that is farther away from the transducer. Note that not only the focal spot can move on the Y plane but also on the X and Z plane.

The delays generated by the delay generator in Figure 2.2 can be calculated using known parameters such as transducer element size, the sound of speed inside of medium, the position of the focus spot. An ultrasound beam alternatively can also be created using a mechanical lens. ([35])

2.2.1. IMAGING VIA ULTRASOUND

Figure 2.3: Figure depicting an ultrasound B-mode image of a fetus inside of the uterus.

To create an image such as a Figure 2.3 a transducer must sweep the beam over a large area of tissue. The ultrasound wave sent by the transducer will encounter different types of impedance from the tissue, this impedance can cause the ultrasound wave to deflect or reflect. This reflection commonly known as an 'echo' is sent back to the transducer where its piezoelectric properties will create a voltage shift. These delays from these echos relative to each other are the inverse of the delays used to create the initial transducer beam. ([33])

An ultrasound wave moves at the speed of sound, as that is a constant the distance of a reflector is known based on the time of the reflection. To make sure that the axial resolution is as high as possible the usage of a short ultrasound pulse is advised as to when a pulse length stretches over multiple Reflectors, the resulting echo will not be able to distinguish between the two reflectors, as can be seen in Figure 2.4. ([34])

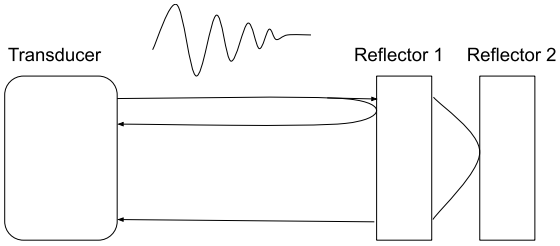


Figure 2.4: This drawing depicts an ultrasound wave produced via a transducer where the wavelength is larger then the distance between Reflector 1 and Reflector 2 resulting in an echo which is unable to distinguish between the two reflectors.

2.2.2. NEUROMODULATION VIA ULTRASOUND

To stimulate a nerve using ultrasound a constant ultrasound wave is necessary, in contrast to the short pulse required for imaging. The beamwidth needs to hit the nerve for an action potential to occur. Multiple factors are influencing the nerve when subjected to ultrasound, these factors can be split up into thermal effects and non-thermal effects. There is not yet a clear scientific consensus on how these factors precisely affect the nerves, however, and as such all of the information surrounding ultrasound, neuromodulation is hypothetical. The effects of ultrasound on the nerve are simulated using the neuronal intramembrane cavitation excitation (NICE) model which is based on the thermal and non-thermal effects which shows comparable results compared to real implementation within human and animal models ([36]).

Normally high-intensity thermal effects induce biological changes including tissue homogenization, protein denaturation, and DNA fragmentation that eventually lead to cellular death. If the thermal energy is kept low enough 3 main thermal effects influence the nerve; Synaptic changes with constant temperatures, rapidly changing temperatures, slow prolonged temperature changes. However, as the amount of thermal energy delivered by ultrasound is very low these effects are considered negligible. ([36–38])

When a low amount of thermal energy is delivered to the nerve over a period it manifests as a perturbation of neuronal levels such as ultrastructural synaptic changes, decrease in synaptic vesicle count, and expansion of pre-and postsynaptic junctions. In other words, neural signaling is blocked between neurons at temperatures below the energy input level which causes the nerve to dissolve.

Slow and prolonged heating of neuronal tissue reversibly inhibits action potential generation and propagation by increasing the rate of sodium channel inactivation and potassium channel activation suppressing the action potential activation. In low intensity focused ultrasound, the thermal effects of sonification can be seen as neglectable, as minimal temperature increase in tissue only is around 0.1-degree Celsius or lower.

The Main non-thermal effects are: Stable cavitation, unstable cavitation, compression

and refraction, and finally acoustic radiation. These effects are seen as the main driving force in ultrasound neuromodulation. ([37, 38])

Stable cavitation occurs when in areas of peak low pressure, gas-filled microbubbles precipitate out of solution and oscillate periodically without damaging the surrounding tissue. Stable cavitation of nanobubbles within the polar membrane of the lipid bilayer of the cell might play a role in action potential generation. These rapid oscillations can cause the spread of the bilayer into two monolayers under negative pressure, while under positive pressure the monolayers are squeezed against one another. The accumulated nanobubbles in the hydrophobic zone during this process modify the bilayer's local curvature altering the excitable cell's overall membrane capacitance changing the membrane capacitance as seen in Figure 2.5.

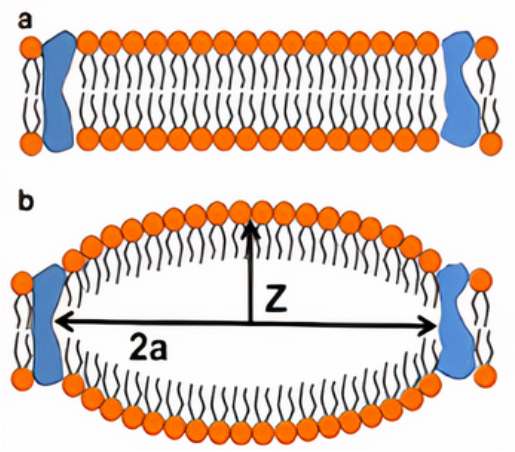


Figure 2.5: Figure depicting the effect of negative pressure on the cell membrane causing the lipid bilayer to split into two monolayers.

Unstable cavitation occurs when exposed to intensities higher than stable cavitation. When this occurs, the microbubbles have the risk of expanding non-linearly and collapsing. This is called inertial cavitation. Depending on the size of the microbubbles and the employed sonication parameters, inertial cavitation can destroy surrounding tissue.

Compression and rarefaction of the surrounding plasma during sonication alters the fluidity and permeability turning the cell layer from the gel phase into the Fluid phase as seen in Figure 2.6 causing changes in the permeability of the cell membrane.

In acoustic radiation, the ultrasonic waves provide steady pressure on the targeted neurons. The mechanical energies conveyed by these ultrasound waves stretch and distort the cell membrane resulting in adjustments of the mechanics of ion channels. These adjustments affect the crossing of the ions through the lipid bilayer and thus the neuron's

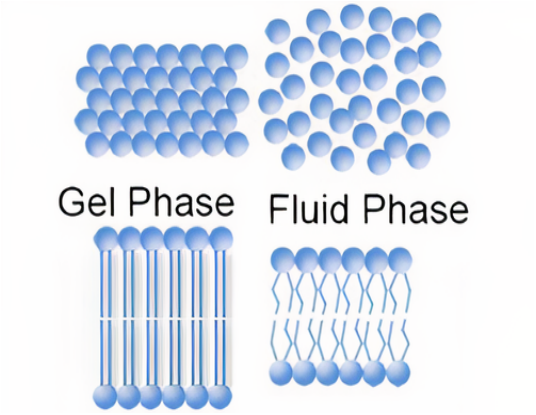


Figure 2.6: The effect of compression and rarefaction of the surrounding plasma during sonification on the cell membrane.

action potential probability.

2.3. REVIEW OF PREVIOUS WORKS

Neuromodulators have been shown to be successful in animal models ([39, 40]), in-vitro ([41]), and Human test subjects ([42, 43]), however none of these works would fall under the terminology of 'wearable' of which the requirements are explained later down the line in this section.
To create a good wearable ultrasound neuromodulator design, inspiration must be taken from previous works, in which both the good and the bad must be recognized, to combine the positives of previous works and leave out the negatives.
First, the definition needs to be made what wearable means in the context of this literature review, as such the following criteria have been established.

- The neuromodulator needs to be able to stimulate the nerve chosen by the wielder.
- The neuromodulator needs to be able to make an ultrasound image of the nerve and surrounding tissue.
- The neuromodulator should be able to freely sweep the target area by forming a beam of ultrasound using a phased ultrasound transducer array.
- The neuromodulator should be able to perform its tasks autonomously.
- The user should not experience any discomfort from wearing the neuromodulator or be perceived as cumbersome.
- The neuromodulator should be able to find the nerve with high accuracy and within a small time frame.

2.3.1. A CMOS 2D TRANSMIT BEAMFORMER

The work of [1] seeks to improve on commercially available ultrasonic transducers for neuromodulation applications, which are typically single focused transducers with a large form factor and off-the-shelf electronics for operation. A CMOS 2D beamformer with integrated lead zirconate titanate (PZT) ultrasonic transducers is reported in this paper for peripheral nerve neuromodulation. Without an acoustic matching layer, the suggested prototype of the work may reach a maximum focal pressure of roughly 100 kPa with a 5 V supply at 0.5 cm depth.

The technical specifications of the design made by [1] consists of CMOS 2d beamformer with an integrated lead zirconate titanate ultrasound transducer array. This device is connected to a computer for control of the focal spot of the neuromodulation beam as shown in Figure 2.7.

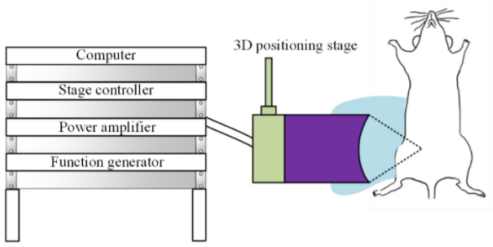


Figure 2.7: Figure from paper by [1]. depicting the setup for ultrasound neuromodulation on an animal model.

The transducer array designed in this work is a monolithic 10 MHz frequency consisting of 676 elements providing 100 KPa at 0.5 cm depth making it suitable for stimulation of the nerve as can be seen Figure 2.8 , however this design is purely focused on neuromodulation as such it is not designed to transmit imaging pulses resulting in no created ultrasound image of the nerve. The work by [1] can beamform via ultrasound delays, this is done via a transceiver circuit that can cause delays for every individual transducer in the transducer array, these delays have a full 360 degree phase coverage with a resolution of 7.5 degrees.

The transceiver circuit is connected to an external laptop, making the only possibility for stimulation external input consisting of the required delays, making this device not autonomous, this combined with the device's inability to create an ultrasound image, the two requirements to find the nerve, as such this device no has nerve finding capabilities. Disconnected from a laptop the overall size of the device is 5 by 4 mm^2, as such the user should not encounter any discomfort from its size.

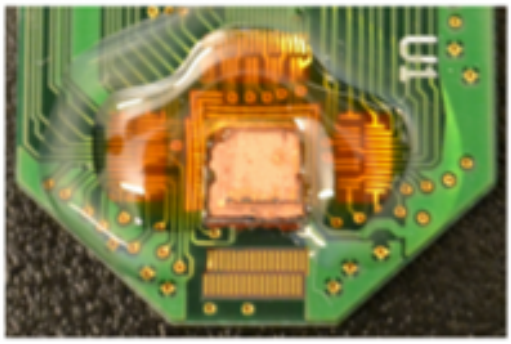


Figure 2.8: Photo taken from paper by [1]. depicting depecting transducer array within CMOS 2d beamformer.

2.3.2. WEARABLE ULTRASOUND FOR IMPROVEMENT MOTOR FUNCTION IN AN MPTP MOUSE

The aim of the work by [2] is to examine whether ultrasound stimulation of the motor cortex can improve parkinsonian motor deficit in a mouse model induced by 1-Methyl-4-phenyl-1,2,3,6-tetrahydropyridine (MPTP). The ultrasound transducer used in this work provides 100 kPa at a depth of 4 mm , this was used to perform low frequency low intensity pulsed ultrasound (LIPUS) 40 minutes a day resulting over 7 days in an indication that LIPUS may be a novel neuromodulation tool for PD treatment.

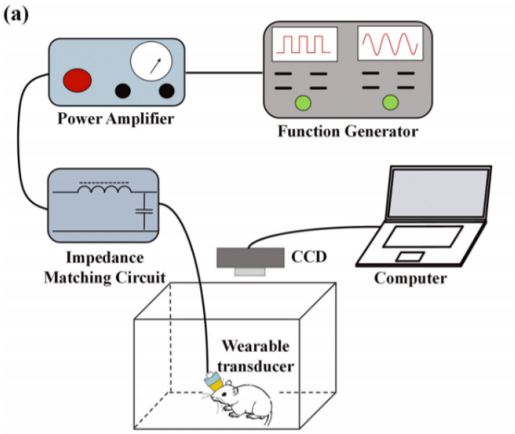


Figure 2.9: Figure from paper by [2]. depicting the setup for ultrasound neuromodulation on an animal model.

The technical specifications of the design made by [2] consists of a single-element focused wearable transducer using PZT-5H/epoxy 1-3 composite as the piezo material

for the transducer due to its high electromechanical coupling factor and low acoustical impedance. This wearable transducer is connected towards a function generator to produce neuromodulation ultrasound waves. This setup was tested on a mouse model altered by an injection of MPTP to see whether neurostimulation on the motor cortex improves a parkinsonian motor deficit as shown in Figure 2.9.
The ultrasound transducer provides 100 kPa at a depth of 4mm making it suitable for neuromodulation, as it constructed with only neuromodulation in mind as such it's not suitable for producing ultrasound imaging pulses, nor does the setup does not have any circuitry to convert the imaging echoes towards digital data. As can be seen from 2.10 the transducer consists of one element pressed into a steel ball at a temperature of 65 degrees to achieve a natural convex shape. This transforms any ultrasound wave into a natural beam with a focus spot, however as the shape of the transducer is set it is impossible to change the focal spot in 2 dimensions unless physically moved.

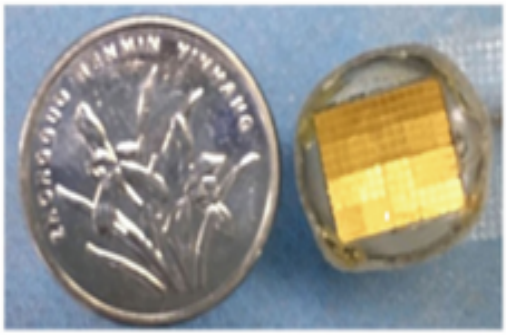


Figure 2.10: Figure from by [2] depicting the setup for ultrasound neuromodulation on an animal model.

As shown in 2.9 the ultrasound transducer is connected to an external function generator, making the only possibility for stimulation external input making it non autonomous, this combined with the lack of capability to find the nerve means the device has no nerve finding capabilities. As the size of the ultrasound transducer is not specified an assumption need to be made, considering the ultrasound transducer fits on a mouse's skull and is smaller than a coin it can be reasonably assumed the ultrasound transducer would fit on a human neck without being seen as cumbersome or otherwise cause discomfort.

2.3.3. ULTRASOUND PATCH FOR IMAGE-GUIDED NEUROMODULATION

This work by [3] describes the design and validation of body-conformal active ultrasound patches for image-guided neural stimulation. A mechanically flexible patch on the probe gives patient-specific feedback on array curvature for real-time ultrasound beamforming focusing optimization. The modulation shows a sensitivity of 80 kPa/V with a 3 MHz bandwidth, while the imaging array has a sensitivity of 20 kPa/V with a 6 MHz bandwidth, according to experimental data from a flexible prototype. Also given is an algorithm for accurate and automated localization of specific nerves utilizing neighboring

blood vessels (e.g., the carotid artery) as image markers.

The technical specifications wearable device created by [3], consists of two transducer arrays. A 64 element linear transducer array is used for imaging and an 8 element phased transducer array is used for low-intensity neuromodulation, both transducers are made from lead zirconate titanate (PZT) with an acoustic matching layer made from acrylic plastic, with an adhesive made from silicon-based materials. A lensing layer of a material matching the acoustic matching layer is added to the imaging array. These transducers are connected to two pre-made chips; a MAX14808, Maxim Integrated to generate eight-channel transmit waveforms, and an eight-channel receiver (AD9276, Analog Devices) to process the resulting echoes. All this circuitry is combined in flexible PCB this PCB is connected with a wire towards an Altera System on a Chip (SoC) which will act as the controller and as the digital backend. For the full overview see Figure 2.11.

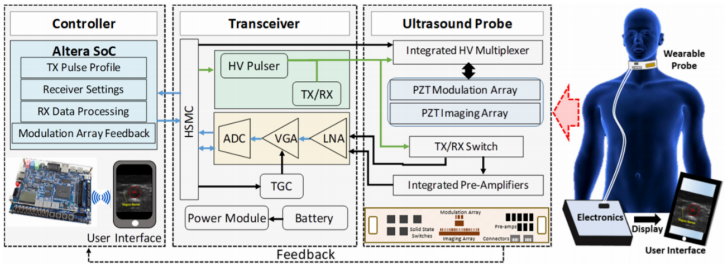


Figure 2.11: Figure from paper by [3]. depicting the setup for a wearable ultrasound neuromodulator.

The neuromodulation provides a pressure of 80 kPa/V making it suitable for Neuromodulation, idem ditto for image generation as a linear array is used to create an ultrasound image from the tissue surrounding the vagus nerve. The neuromodulation ultrasound beam is created via ultrasound delays, since the neuromodulation transducer array is a phased type that uses a delay circuit to create a neuromodulation focus beam with an estimated spatial resolution of 1.2 mm. These delays are calculated using an Altera SOC. This Altera SOC is the controller in this device as it controls both the beamforming as well as searching for the vagus nerve. Using feedback from the PCB sensors, adjustments are made for the aforementioned functionality, this makes this device autonomous as no outside commands or interference is required for the device to function. As the device consists of a flexible PCB collar wrapped around the neck with a size of approximately 4.5 cm by 25 cm, connected via a wire towards a suitcase filled with the Altera SoC. The size and complexity impede the wearer and causes discomfort.

To find the vagus nerve using an ultrasound image created by the imaging array a template matching algorithm is used. The images used for template matching will first go through the following steps before the cross-correlation factor is calculated. ([44])

- The target image goes through a gaussian filter to smooth out the edges and remove any noise.

- It is then passed through other pre-processing filters like dilation, noise pixel removal, thresholding, etc.

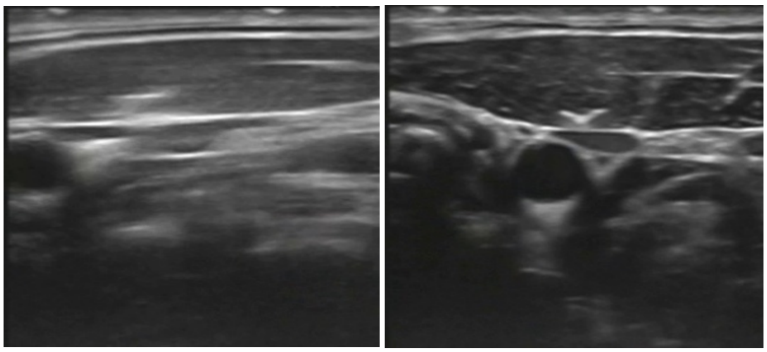


Figure 2.12: Two images taken from [4] showing a completely different shaped Vagus nerve within an ultrasound B-mode image.

Then template matching is applied to compare the target image with all the stored templates which correspond to different orientations. In template matching the algorithm loops through every pixel of the target image to compare that pixel and its surrounding pixels to the template. The set threshold determines to what level the pixels in the template have to match the considered area in the frame for the area to be identified as a marker location. This threshold ranging from 0 to 1 thus determines the precision of the system, 0 meaning no pixels match the template, 1 meaning all pixels have to match the template image. The full formula is described in more detail within section 3.2.1. This technique does have its drawbacks in implementation, one of which is the accuracy, the templates are static, and the vagus nerve is dynamic since it moves around, stretches, changes shape, or otherwise distorts (see Figure 2.12 a,b). This means that a picture of the template will not always be a good representation. The second problem lies in the large number of calculations that need to be done, e.g., if a target image of 720×480 pixels is cross-correlated with a template image of 70x70 pixels this translates into a comparison of 1693440000 pixel pairs, combined with the multiple templates, the calculations will take a large time on a microcontroller or an SoC. This leaves the nerve finding capabilities of the template matching algorithm within this work slow and inaccurate but usable.

2.3.4. SUMMARY

As can be gleaned from table 2.1 there is not a single implementation that satisfies all of the criteria presented. This raises the following problem: “How do we implement a wearable real-time autonomous ultrasound neuromodulation device?”.

Work	Neuromodulation	Imaging	Beamforming	Autonomous	Cumbersome	Quick and precise
[1]	yes	no	yes	no	no	no
[2]	yes	no	no	no	no	no
[3]	yes	yes	yes	yes	yes	no

Table 2.1: Table containing works from [1], [2], [3], with an overview of how they compare against the requirements set in section 2.3, note that none of the featured works satisfy all requirements.

2.4. PROPOSED APPROACHES

To satisfy the requirements to make a wearable ultrasound neuromodulator in Figure 2.13 a design is proposed. In this design, a flexible wearable patch is introduced which can be applied by the patient simply by pressing it against the left side of the neck around the area of the jugular vein. From this point onwards the patch can be turned on and the neuromodulator will start scanning for the vagus nerve autonomously and once found will start exciting the nerve on a frequency proposed by the user. While exciting every two seconds another scan will be done to search for the location of the vagus nerve to make sure that it is still excited in case of, for example, movement. As can be seen from this picture it is smaller than any of the previously made ultrasound devices while advancing on functionality. To do this there are two designs proposed, while both designs are the same input, output, and dataflow, they both have their unique challenges associated with them. Do note that further within the confines of this thesis the focus has only been laid on implementing the vagus nerve finding capability is detailed within both designs and not implementing the whole design detailed within these sections, as such all components aside from the nerve finding capability is should be considered future work and within the following sections meant to give a more realistic interpretation of the wearable environment that the object detection will operate in.

2.4.1. VAGUS NERVE FINDER USING ANDROID

In Figure 2.14 the cross-section of the first proposed solution is given, which for ease of writing will be referred to as solution 1. The body of the device will consist of a flexible pad with an adhesive sticky ultrasound gel. On top of this patch, a battery will protrude which can be replaced if necessary. The control of the entire neuromodulator will be done via a microcontroller that has either Wi-Fi or Bluetooth access. This microcontroller will contain the predetermined delay values needed for a complete imaging sweep of the tissue surrounding the vagus nerve while also being able to calculate the delays necessary for a neuromodulation ultrasound beam. This information will be sent via I/O ports towards the transceiver chip where it will be converted into pulses with the delays ordered by the microcontroller. The microcontroller will also be receiving information from the transceiver regarding the echos created by the imaging. These echos will be converted into a complete image by the microcontroller which will be sent via either Direct Wi-Fi or Bluetooth towards the patient's Smartphone.
The image sent towards the patient's smartphone will be put through a neural network algorithm using the Smartphone graphics card. Once the position of the vagus nerve has been located with a degree of certainty the angle and depth will be calculated in relation to the transducer, the angle and depth will be sent back towards the microcontroller. This

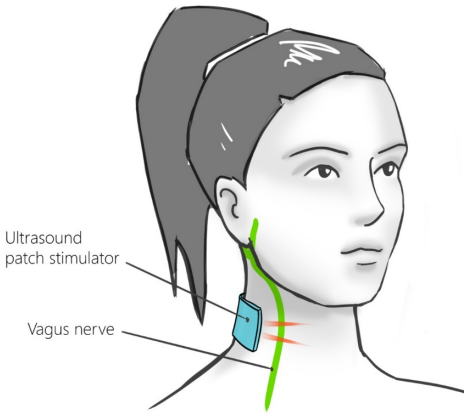


Figure 2.13: Figure depicting proposed ultrasound setup of wearable ultrasound neuromodulation patch.

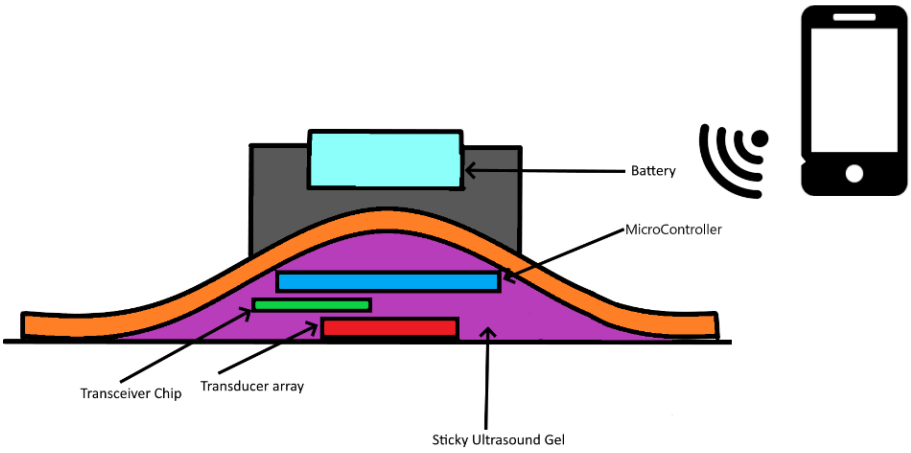


Figure 2.14: Figure depicting the cross-section of the mobile phone based ultrasound wearable neuromodulation patch.

will be the input for calculating the delays. The transducer chip will be connected with the transceiver chip and will be able to send both imagine waves as well as neuromodu-

lation waves. The transceiver chip will handle both the bit data from the microcontroller concerning the delays and convert them into pulses with delays matching the bit data, and they will handle the echos created by imaging and convert them into digital data which can be handled by the microcontroller.

The main problem with solution 1 lies in the connection between the phone and the microcontroller, as there is a large amount of information sent wirelessly between two devices which due to its medical nature cannot be corrupted nor can it be encrypted with loss. This causes both problems with the speed of this connection as well as cybersecurity. The speed of the connection will be a problem since the neural network algorithm needs to have all the bits of the image before it can proceed to estimate the position of the vagus nerve, this might cause a large bottleneck to appear to in case the image uses a higher resolution.

The second problem concerning the wireless connection between phone and microcontroller is cybersecurity. An open unencrypted wireless data stream leaves itself easily open for a man-in-the-middle attack. During a man-in-the-middle attack, a bad actor will intercept the transmission between microcontroller and smartphone or vice versa and corrupt to possibly harm the patient. Fears of hackers targeting electrical medical devices are not unknown, as famously former Vice President Dick Cheney had a wireless connection towards his pacemaker removed This can be solved via encryption, Public Key Pair Based Authentication, or Strong login credentials, all of these solutions will increase the size of the send data and it will add calculations used for verification on both the microcontroller- and smartphone-end further increasing the time needed to send data.

The third drawback might be the inclusion of laws surrounding medical data being used on smartphones, as throughout the world different rules exist and can change leading to an unworkable product down the bottom line.

The final drawback is concerning the availability of smartphones being able to run neural networks using a graphics card. According to android developer studio, the minimum SDK required for neural networks to easily connect to the smartphone's graphics card is installed on 60.8 percent of Android devices as shown inside android studio. As of 2020, there are 3.8 billion smartphone users over the world, considering Android has a market share of 72.48 percent as of December 2020 [45], a rough estimation can be made of eligibility of 1.674 billion people out of a worldwide population of 7.842 billion, this results in around roughly 23 percent of the worldwide population being able to use this design for a neuromodulator device. Most of these users are inside richer countries which brings ethical concerns towards distribution of health care. The goal should be for healthcare to be affordable and easily distributed for all, this design brings this goal into question.

2.4.2. VAGUS NERVE FINDER USING FPGA

The second proposed solution, referred to within this thesis as solution 2 is very similar to proposed solution 1, as can be seen in Figure 2.15. The microcontroller, transceiver, and transducer all have the same functionality. The main difference is how the functionality used by the smartphone is overtaken by an FPGA which eventually further into its development will be transformed into an ASIC, however due to the constraints that

master's thesis brings within this thesis only an FPGA will be implemented. This FPGA will be referred to as the 'Imaging Algorithm Chip'. The connection between the Imaging Algorithm Chip and the MicroController will be done via IO ports. An alternate option is to use a pre-made ASIC for computer vision, both options have their positives and negatives.

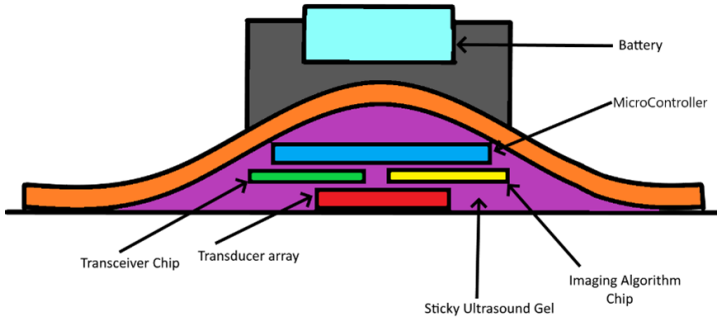


Figure 2.15: Figure depicting the cross-section of the 'Imaging Algorithm Chip' based ultrasound wearable neuromodulation patch.

For the FPGA prototype use case a pipeline will be created which will convert pictures taken of the vagus nerve into an HDL representation of the imaging algorithm/neural network. In theory, this will provide a unique Neural network or algorithm per patient, this should provide a higher accuracy, and faster inference compared to generalized imaging algorithms/neural networks. However, developing the prototype on an FPGA will take a long time, which is only the first step, as this prototype, once it has proven to results in adequate performance, the conversion from an FPGA prototype to an ASIC will cause further time as well as money, however once a design is created producing the ASIC should prove cheaper than buying a pre-made ASIC depending on how high the markup is.

In the use case of a pre-made ASIC, a neural network or imaging algorithm will be implemented on an ASIC specifically made to run computer vision tasks. Implementing computer vision task should prove to be easier then creating a FPGA prototype as ASICS usually work straight out of the box and only the Software part needs to be implemented. However, a generalised ASIC will never provide as fast of an inference, accuracy, and power efficiency then a custom-made design for a specific task. Due to this reason the choice is made to focus on implementing an FPGA instead.
The only difference between the smartphone and the Smartphone implementation will be the output and the connection between the microcontroller. The Smartphone will output the angle and the depth, while the Imaging Algorithm Chip outputs the pixel location of the input image, leaving the microcontroller to calculate the angle and the depth.

Solution 2 in theory will solve a lot of problems regarding both the bottleneck in speed as well as cybersecurity issues, as the wireless connection between the microcontroller and the smartphone is replaced by a wired I/O connection between the microcontroller and the Imaging Algorithm Chip.
As it is a standalone product, no mobile phone is needed, increasing the userbase by a large degree. Due to the inclusion of either an FPGA or an ASIC the basic price will be higher compared to solution 1, the severity of this price increase is unknown and will depend on the complexity of the HDL design of the Neural network/imaging algorithm. As one of the main goals of the medical device is still to provide accessible and cheap healthcare the choice of which proposed solution will be better will depend to a large degree on the complexity of the Neural network/imaging algorithm.

2.4.3. MOTIVATION FOR APPROACHES

In Figure 2.16 the class diagram of the approach is described as both approaches 1 and 2 have the same general data flow they can be combined into the same diagram. The main characteristic that sets this implementation apart from the previous designs is the usage of a GPU, FPGA, Custom Made Chip to find the Vagus nerve, the usage of these devices was chosen to improve upon the time required to find the Vagus Nerve. As imaging algorithms/Neural networks are considered "embarrassingly parallel" ([46]) the usage of a high core card like a GPU or FPGA should improve the speed radically. In an embarrassingly parallel task, there is little, or no effort needed to separate the problem into several parallel tasks, in the use-case of finding the vagus nerve these parallel tasks can be the calculations done over a pixel or a bundle of several pixels.

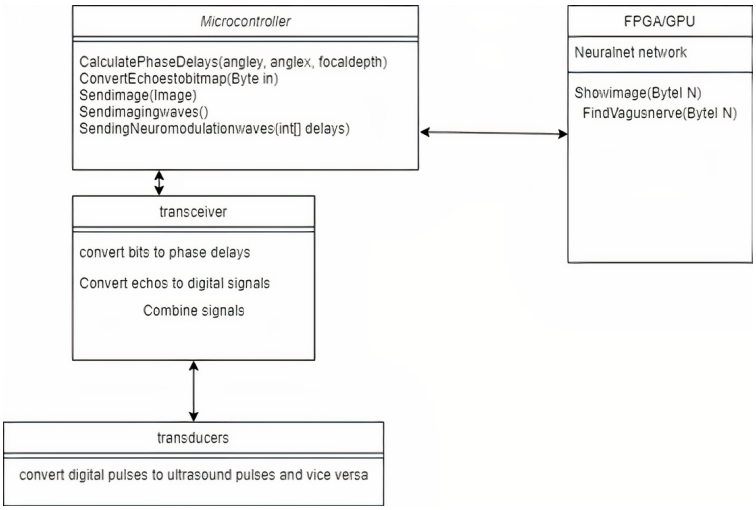


Figure 2.16: Class diagram depicting the general dataflow of ultrasound wearable neuromodulation patch.

The choice to use a Neural network or a more dynamic imaging algorithm compared to

template matching is its more dynamic nature, as a vagus nerve moves around, stretches, changes shape, or otherwise distorts (see Figure 2.16 a,b). A more dynamic algorithm should be able to handle any drastic changes in the ultrasound image without losing too much accuracy.

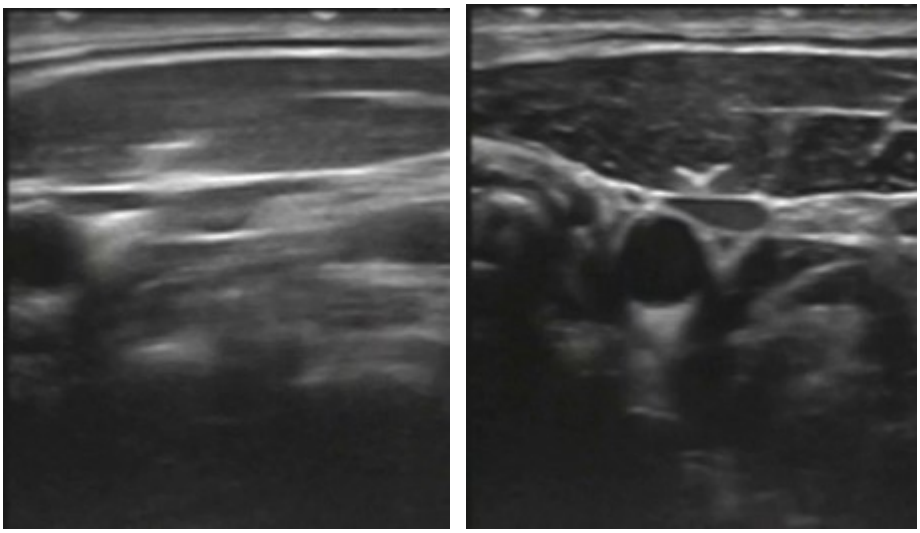


Figure 2.17: Two images taken from [4] showing a completely different shaped Vagus nerve within an ultrasound B-mode image.

Using a small microcontroller compared to a system on a chip is motivated by the fact that all of the calculations needed to find the vagus nerve are done by GPU/FPGA/Custom Made Chip. The previous iteration by [3], chose a system on a chip for its high clock speed. While this has merit since speed is an important factor the usage of multicore card makes it obsolete, the calculations that are leftover such as phased array delay calculations can be considered negligible in terms of recourse requirements, as such to make the device as small as possible the usage of a microcontroller was chosen.
The choice to have a transducer array that can do both imaging, as well as neuromodulation in contrast to two separate transducer arrays that specialize in either imaging or neuromodulation, is based on works showing that transducer arrays which are capable of both imaging and neuromodulation can do so without too large of a drawback to resolution or focal point pressure. ([47]). The transceiver circuit will be based on the phase modulator design by [1], considering its small size and high resolution compared to premade boards used by e.g., [3].

Based on the two designs discussed within these pages a following chapter is introduced which compromises the general background info and methodology surrounding various facets of implementing an wearable object detector of the vagus nerve using either an FPGA or a Android Phone using a GPU.

3

METHODS

3.1. INTRODUCTION

THIS chapter compromises the general background info and methodology surrounding various facets of implementing an wearable object detector of the vagus nerve, these facets can be subdivided under the following aspects: background info surrounding computer vision techniques: Template Matching, and Artificial Neural Networks. Background Motivation regarding certain design choices: Amdahl's Law and Quantisation. Methods chosen for validation of every implemntation created within this thesis project: Image dataset, and accuracy measurement method. Finally background information is given surrounding prototyping platform: The Ultra96V2.

3.2. OBJECT DETECTION METHODS TO FIND THE VAGUS NERVE

Based on section 2.4.3 two methods were chosen to find the vagus nerve, Artificial Neural Networks and template matching. Both of these implementations are configured to produce a bounding box which is centered on the vagus nerve. Template matching has been chosen to create an implementation which builds on top of the progress made by [3] by furthering inference using the proposed designs in section 2.4. Artificial Neural Networks should improve the accuracy of detecting the dynamically shaped vagus nerve. This is not to say that Artificial Neural Networks and Template matching are the only methods of object detection, but for the sake of keeping the thesis manageable, these two methods have been chosen. Methods of object detection not implemented within this thesis such as Block matching can be considered as future work.

3.2.1. TEMPLATE MATCHING

Template matching can be seen as a very basic form of object detection. Using template matching, objects can be detected in an input image using a "template" containing the object to be detected.
This requires two inputs:

1. Source image: e.g. a b-mode ultrasound image of the vagus nerve and surrounding tissue, in this image locating the vagus nerve is the objective.
2. Template image: e.g. a b-mode ultrasound image of the vagus nerve, this is used to locate the vagus nerve inside the source image

To find the vagus nerve the template image is slid across the source image as shown Figure 3.1

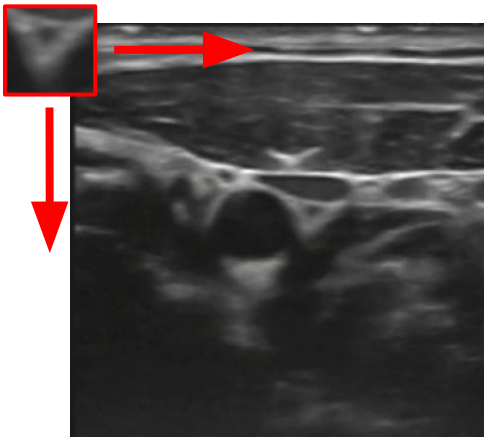


Figure 3.1: Figure showing the sliding of the template over the target image.

At every pixel within the source image a score is calculated between 0 and 1 representing the similarity between template image and the source image at that pixel's location. This score is calculated as

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{3.1}$$

With T being the template image and I the source image.

3.2.2. ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANNs) are statistical models that are partially inspired by and modelled after Biological Neural Networks (BNNs). Nonlinear correlation between distinct inputs and outputs can be computed using ANNs. They are powerful instruments that can assist in the solution of problems that cannot be handled using an algorithmic approach. Neural networks can be used to handle problems like image categorization and natural language recognition. A full detailed explanation of the general working of ANNs is detailed within appendix A, this is recommended reading if the reader is not familiar with the workings of neural networks or wishes to find a deeper explanation regarding a specific subject mentioned within this thesis. For ease of reading within this thesis Neural Networks (NNs) refer to Artificial Neural Networks.

3.3. DETAILS REGARDING IMAGE DATASET USED WITHIN THESIS

The image dataset consists of 199 images taken from the video provided by [4]. Within this video the patient turns his head and swallows, giving some variety to the placement and deformation of the vagus nerve. Every 4 frames within the video an image was retrieved with the surrounding Graphics User Interface from the ultrasound machine and external Point of view cropped out. Within all these images the vagus nerve and surrounding tissue were annotated with the centre of the annotation box of variable size being centered square on top of the vagus nerve.
The surrounding tissue triangle shape makes a distinctive feature which can be more easily recognized via algorithms under the assumption being that the ultrasound neuromodulation beam will always be point squarely in the middle of the annotation box.
All images within the dataset contain a single annotation of the vagus nerve.
These images where annotated in the YOLO Darknet text format. The dataset complete with annotations can be found at [48].

3.3.1. DATASET TRANSFORMATIONS

The bounding box placement shows very little variation as can be seen in Figure 3.2.

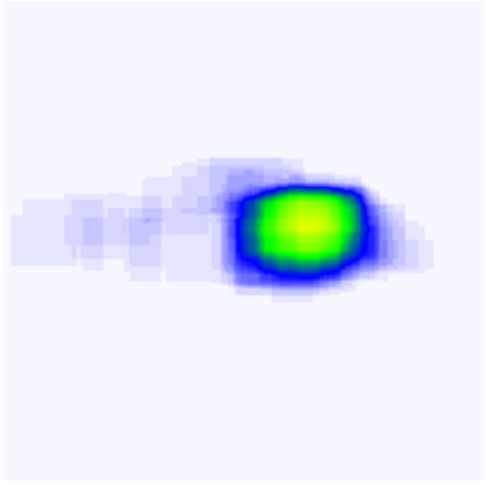


Figure 3.2: Heatmap showing the position of every annotation within the dataset used in this Thesis.

To increase the variety but keep within the bounds of realism the following transforms were applied individually up to 3 times to every image within the dataset resulting up to a maximum of 3 new images for every existing image.

1. Rotation: Between -16° and +16°
2. Bounding Box Shear: ±12° Horizontal, ±12° Vertical

The rotation simulates different angles under which the ultrasound probe is applied to the targeted area and the Bounding box shear simulates compression of the targeted tissue.
Finally the dataset is split randomly into 70% training, and 30% to the test set in order to protect against overfitting.

3.3.2. ACCURACY MEASUREMENT USED IN VERIFICATION IMPLEMENTATIONS

All of the imaging algorithms described in the coming sections return bounding boxes. These bounding boxes represent the predicted position of the vagus nerve.
As described in section 3.3 the dataset does not contain any false positives, this means that the accuracy measurement is purely taken in precision and not in recall.
In order to measure the accuracy of the implementations used, a 2-part function has been defined.

$$0.95 * 1_{ij}^{oj}[(x-\hat{x})^2 + (y-\hat{y})^2 \tag{3.2}$$

The term in formula 3.2 consists of the bounding box, indicated by the x and y coordinates parametrized with an offset of a particular grid cell location. That secures that the coordinates are bounded between 0 and 1. A scalar is set at 0.95, accuracy of the bounding box is seen as critical, it is crucial that the centre of the bounding box comes as close to the vagus nerve as possible.

$$0.05 * 1_{ij}^{o}[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \tag{3.3}$$

In the second term bounding box width and height are normalized by the image width and height so that they fall between 0 and 1, as small deviations in large boxes matter less than in small boxes. square root of the bounding box width w and height h instead of the width and height directly to partially address this problem. A scaler is set to 0.05 as the size of the box is not relevant to its centre position.

3.3.3. EFFECT OF DATASET ON DESIGN OF IMPLEMENTATION

As can be seen within Figure 3.2 the dataset can be considered flawed, as such there is a realistic scenario that a generated NN might be needed to be retrained, it might even be shown that the size of some of the layers within the created NN can be considered lacking leading to decreased accuracy.
In this case coding an implementation of a NN for use within an FPGA in HDL requires rewriting of the code, for an untrained user this could take some time.
As such the choice was made to create an automated pipeline in which the user only has to make adjustments to a NN and the end output would be an HDL representation and/or bitstream for use within an FPGA.
To achieve this aim the choice has been made to use existing open source frameworks both with wildly varying ideas of how a neural network should be implemented.
The choice has been made to use FINN and NNgen with arguments in favour and against detailed within section 4.6 and section 4.5 respectfully. A small sub aim of this thesis is now to create a pipeline in which future users will have to interface as little as possible with any kind of code or parameters to gain an as accurate and as efficient as possible implementation within an FPGA of a NN.

3.4. HARDWARE USED FOR IMPLEMENTATION

The Ultra96V2 is an ARM-based, Xilinx Zynq UltraScale+ MPSoC development board based on the Linaro 96Boards specification, the Ultra96-V2 boots from a 16 GB microSD card. The xczu3eg-sbva484-1-I fpga part contained inside is considered to be the main platform for running accelerators to find the Vagus Nerve.
The specs that are given for the Ultra96V2 are detailed within 3.1

Display Name	Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Min Operating Voltage (V)
Ultra96-V2	xczu3eg-sbva484-1-i	484	82	70560	141120	216	0	360	0.825

Table 3.1: Specifications given for the Ultra96V2 taken from [9].

As the xczu3eg-sbva484-1-I does not contain many resources it is important for every implementation to be as resource efficient as possible, however since this is an objective of the final implementation anyways the xczu3eg-sbva484-1-I could be seen as a proper prototyping platform for the goal of implementing a low resource, low power, high inference object detector for the vagus nerve.

3.5. METHODS OF OPTIMIZING HARDWARE IMPLEMENTATIONS

To implement the methods described within section 3.2 within a hardware implementation such as an FPGA, while getting optimal results such as high inference, high accuracy, low resource usage, and low power usage, two methods have been implemented, Parallelisation and quantisation. Within this section the workings and the advantages of these methods are explained, shown, and proven.

3.5.1. PARALLELISATION

In 1967, Gene Amdahl proposed a scaling law that is often overlooked: sequential computations of a program severely limit the maximum possible acceleration. This means that any non-parallel execution or intercore connection, regardless of the number of additional computing resources, quickly reduces the scalability of parallel applications [49].

The addition of these resources also shows within the power efficiency of accelerators in completing tasks. Within table 3.2 an overview is given of the research by Bonamy et al. their setup containing a Xilinx ML550 board, containing a microblaze core and an FPGA. Within this setup an implementation within the microblaze core referred to as Soft, a sequential implementation of the hardware task on the FPGA represented as HardS, and HardP representing the best parallelised solution in terms of time within the FPGA. As shown in table 3.2 in all assignments that an increase of parallelism lowers the amount of energy usage as well time taken to complete the assignment.

This transfers well to the choice of hardware used (FPGA and GPU) as they allow a larger amount of parallelism than a conventional CPU. A full detailed explanation of

	Matrix mult		Full Search		Deblock. filter	
	Time	Energy	Time	Energy	Time	Energy
Soft (ms, mJ)	10.75	244.79	0.4786	18.2	0.5742	26.09
HardS (ms, mJ)	1.04	61.99	0.0369	2.01	0.0529	3.68
HardP (ms, mJ)	0.38	27.48	0.0246	1.05	0.0417	2.74
Soft/HardP **Ratio**	**28.29**	**8.91**	**19.37**	**17.33**	**13.77**	**9.52**
HardS/HardP **Ratio**	**2.74**	**2.26**	**1.50**	**1.91**	**1.26**	**1.34**

Table 3.2: Table taken from the works of [10] showing an increase of parallelism lowers the amount of energy usage as well time taken to complete the assignment.

the general working of parallelism of NN within FPGA accelerators is detailed within appendix B, this is recommended reading if the reader is not familiar with the various types of parallelism of NNs withing FPGA accelerators or wishes to find a deeper explanation regarding the specific types of parallelism mentioned within this thesis.

3.5.2. QUANTISATION

NN models for object detection tasks can use millions up to billions of floating point parameters. As can be seen in section 4.2, Vivo Las Vagus, the NN designed in the span of this thesis already includes up to a million floating point parameters. This large number of floating point parameters lead to high compute and resource challenges. This makes implementing an object detector NN nontrivial in devices such as mobile phones, FPGAs or ASICs who have limited compute, memory, and power budgets.
Creating a more efficient NN can be roughly broken up in two categories.

1. Creating a more efficient NN Architecture
2. Quantizing the parameters from a floating point representation to a integer representation

Creating a more efficient NN architecture refers to simple guidelines for a designer to keep in mind. Examples of this are choosing kernel sizes and total amount of kernels that lead to efficient RAM usage [50]. Depth wise separable convolutions in contrast to the usual dense convolutions are another good way of lowering the number of parameters needed within a calculation as explained in section D.0.5.

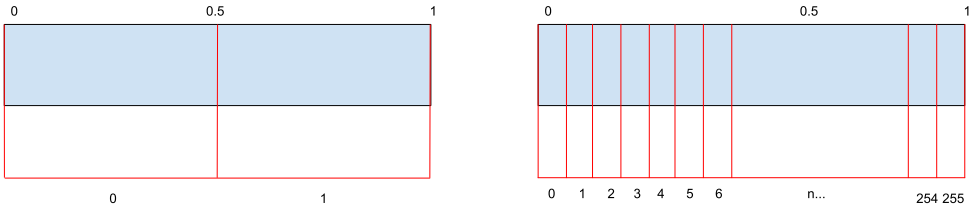


Figure 3.3: A Figure depicting a visual representation of both 1 bit and 8 bit quantisation of a float valued between 0 and 1.

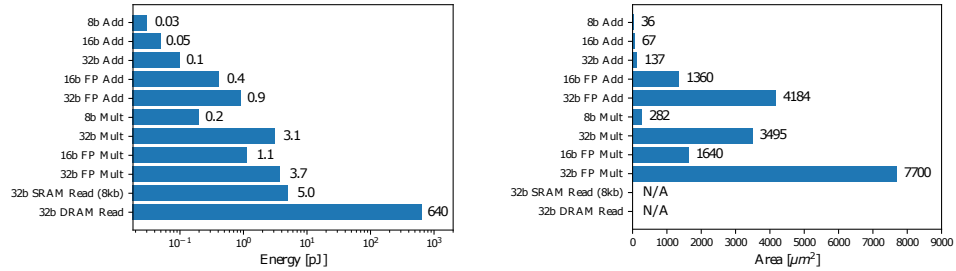


Figure 3.4: Energy and area comparison of different operations for different precision of operands in 45nm technology taken from [5] [6], with the image taken from [7].

Within quantisation a floating point representation of a value to a discrete set of integer values within a specific range dictated by the number of bits used to represent the quantized value. For example a 0 to 1 floating point number might be represented by an 8 bit integer as a number of 0 to 255, or a 1 bit integer as a value of either 0 or 1 as detailed within Figure 3.3.
There are roughly two methods for quantisation: the method of quantizing a trained model with a floating point numbers for parameters into integers (Post-Training Quantisation) and the method of training while quantizing (Quantisation-Aware Training).
The advantage of a quantized architecture is that the computations are expressed in lower precision arithmetic, which is more power and area efficient and faster than floating point arithmetic. Figure 3.4 a and Figure 3.4b visualise the energy and area costs of various operations in 45nm technology respectively implemented on an ASIC.
As can be gleaned from the table all integer operations use less resources and energy then their floating point equivalent, as well as a decrease of resource usage and energy usage when lowering the overall bit size. These smaller sizes allow the designer to implement stored values within smaller, often faster, types of memory 'closer' to the compute logic.

Using the methods discussed within this chapter implementations for wearable object detection of the vagus nerve are be created as discussed within the following pages.

4

IMPLEMENTATIONS

4.1. INTRODUCTION

WITHIN this chapter severally implementations for wearable object detection of the vagus nerve within an ultrasound b-mode image are detailed. The implementations are based on the android phone neuromodulator design in which an android phone is used to find the vagus nerve, and the FPGA based design using an FPGA for finding of the vagus the nerve which for future work will be replaced by an ASIC. For both of these designs a more detailed overview is specified in paragraph 2.4. These implementations are based on the object detection methods detailed in paragraph 3.2, mainly template matching and Neural Networks. Using Neural Networks an object detector is created called Vivo las Vagus, this object detector together with template matching implemented in both an Android smartphone environment as well as a FPGA. Within the FPGA environment there are two implementations of Vivo las Vagus, one created within a systolic array accelerator and one within a streaming dataflow accelerator, both with their own positives and negatives discussed within their own respective chapters.

4.2. VIVO LAS VAGUS: AN OBJECT DETECTOR CREATED USING NEURAL NETWORKS

Vivo las Vagus (VLV) is a single-stage object detection model and was developed during this thesis project. VLV is a single neural network that predicts bounding boxes and class probabilities directly from full ultrasound B-mode images in one evaluation, as such it can be optimized end-to-end directly on detection performance. Object detection is implemented within a loss function in which the loss is calculated based on spatially separated bounding boxes and associated class probabilities based on the works of [51]. The network uses features from the entire image to predict each bounding box, with every bounding box regardless of the class being found simultaneously, his means the network makes decisions about object detection based on the full image and all features within the image.

The input layer takes 224 x 224 RGB images as input. The input image gets divided into a 7 by 7 grid. If the centre of the vagus nerve is located within one of these grid cells, that grid cell will be responsible for detecting it. Each grid cell predicts two bounding boxes with confidence scores attached. These confidence scores range from 0 to 1 and reflect how confident the neural network is whether either three objects are detected (ie. Vagus nerve, Common Carotid Artery, Jugular vein) however in practice VLV is only trained on one object (Vagus nerve). These bounding boxes and confidence scores get combined resulting in the position of the Vagus nerve as shown in Figure 4.1

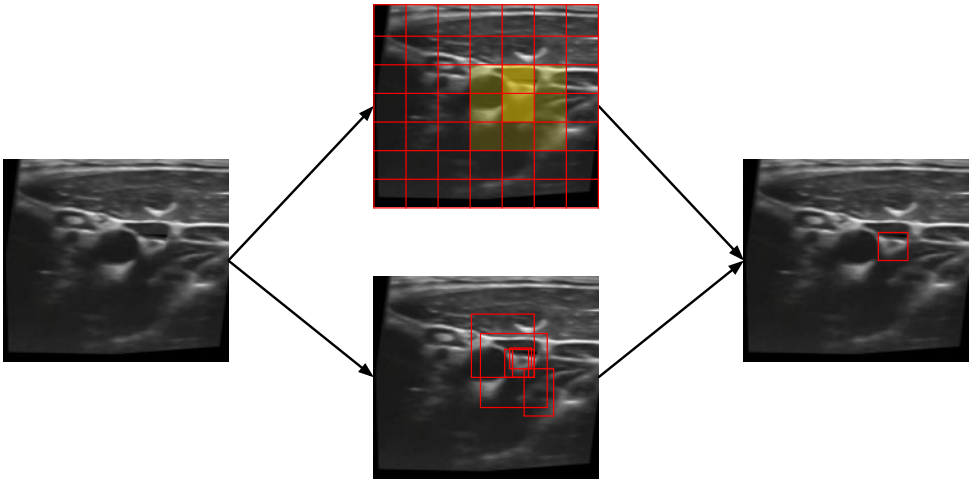


Figure 4.1: Figure depicting both outputs per grid cell(The output boxes and the scores) being combined into a final output.

4.2.1. ARCHITECTURE

Within Figure 4.2 a detailed layer by layer anatomy of the model is given.

The architecture consists of 10 layers consisting of 2d convolutional layers in tandem with RELU layers with 2DMax pool layers intertwined. These maxpool images shrink the overall image by half allowing the grid cells to form. The convolutional layers have a low number of filters due to the need to fit inside of a dedicated accelerator within an FPGA and the goal of only finding one class (the vagus nerve). Due to the lower complexity of the dataset from its greyscaled nature a lower number of filters can be implemented with a lesser risk of losing accuracy.

Almost every convolutional layer passes the requirement set for depthwise convolution: $out\_channels == K * in\_channels$ where K equals a positive integer. As described in paragraph D.0.5 depth wise convolution use a lesser number of parameters compared to standard dense convolutions leading to lower resource usage and higher speed.

For training purposes a dataloader has been created with a batch size of 64 randomised images loaded in from the training dataset every training epoch.

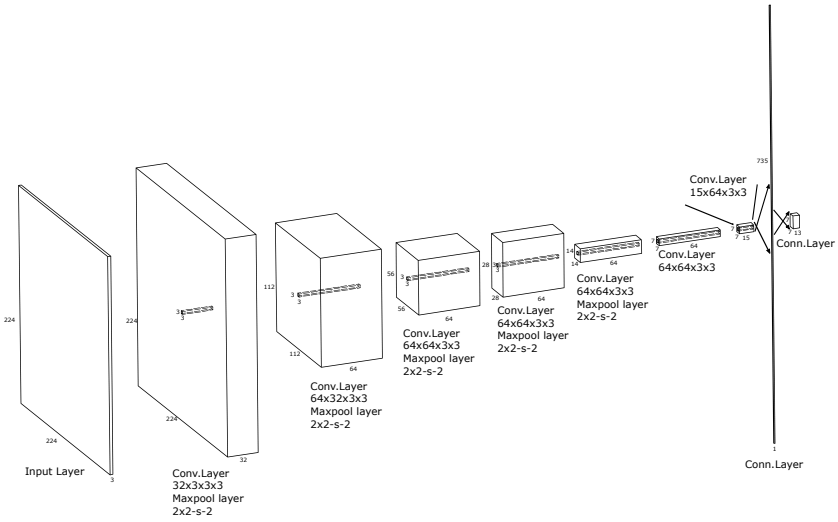


Figure 4.2: Figure depicting an detailed overview of the architecture of VLV.

4.2.2. LOSS CALCULATION

The Accuracy loss function is based on the loss function defined in the YoloV1 paper[51] as such it will be referred to as the *YOLOLoss* function. The *YOLOLoss* function is a combination of the following functions

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \tag{4.1}$$

Function 4.2:The x and y coordinates of the bounding box are parametrized to be offsets of a specific grid cell position, therefore they are also bound to 0 and 1. And only when there is an object is the sum of square error (SSE) estimated.

$$lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{4.2}$$

Function 4.3:The width and height of the bounding box are normalized by the width

and height of the image to be between 0 and 1. SSE is only estimated when there are objects. Because the small deviations in large boxes are smaller than those in small boxes. The square root of the bounding box width w and height h instead of the width and height directly to partially solves the problem.

$$\sum_{i=0}^{S^2}\sum_{j=0}^{B} 1_{ij}^{obj}(C_i - \hat{C}_i)^2 \tag{4.3}$$

$$\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B} 1_{ij}^{noobj}\,(C_i - \hat{C}_i)^2 \tag{4.4}$$

Function 4.3 and 4.4: Many grid cells in every image do not include any objects. This causes the model to become unstable by pushing the *confidence* scores of those cells towards zero, frequently overpowering the gradient from cells that actually contain objects. As a result, the loss from confidence predictions for boxes without objects is reduced, with noobj=0.5.

$$\sum_{i=0}^{S^2} 1_{i}^{obj} \sum_{c\in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{4.5}$$

Function 4.5: When there are objects, SSE of class probabilities

4.3. IMPLEMENTING THE OBJECT DETECTORS WITHIN AN ANDROID PHONE

The implementation of an object detector using a microcontroller and mobile phone as detailed within paragraph 4.3 can be broken up within two separate subjects: Transferring a B-mode ultrasound image from the microcontroller within the Ultrasound patch towards the android phone and implementing the object detectors within an android phone, all of the subjects are described within the following chapters.

4.3.1. TRANSFERRING AN ULTRASOUND IMAGE FROM MICROCONTROLLER TO ANDROID PHONE

The main problem with implementing the microcontroller-mobile phone model described in paragraph x is to transfer the data of the generated B-mode ultrasound image from the microcontroller towards the Mobile phone without losing any data possibly corrupting the accuracy.

As such a lossless protocol has been implemented using a microcontroller and an android phone using Bluetooth. As Bluetooth is a low power implementation and the range between the users mobile phone and ultrasound patch is expected to be short, it would offer the ideal tradeoff between battery life and range vs for example direct Wi-Fi.

Another large proponent of Bluetooth is that it is one of the worldwide most implemented wireless protocols allowing users around the world from a large variety of economic backgrounds to use the wireless patch.

To create a prototyping environment first the choice of a microcontroller had to be taken, based on previous research the choice was made to include the tiny pico V2 within this test set up. To create a connection between the mobile phone for sharing the images

code for both the android phone as well as the Tinypico is needed as described in Figure 4.3. A connection was made using the Bluetooth module *SerialBT* in which a Bluetooth profile is created. This profile will then start advertising itself for a connection with a client, once this profile has a client (the android phone) it will be open for input from the client.
Once an arbitrary command is given by the client, in this case an 8 bit Uint representing the number 46, a quantized 224 by 224 pixel quantized to 8 bit Uint b-mode ultrasound is send in packages of 8 bit, within [52] the code for the microcontroller is written.
From the perspective of the android phone a scan is done for the created profile using the *BluetoothAdapter* module with android. Once a connection the user will be able to send the *send image* command by pressing a button, this will send a 8 bit Uint number of 46 via the *outputstream*.
Have these requirements been fulfilled the BluetoothAdapter will open its input stream via the *getInputStream* command. This input stream will be read as 8 bit Uint and all the values taken from this input stream are written towards a 2D int array. This 2d array will be the basis for vagus nerve detection, code for the android side is written in [52].

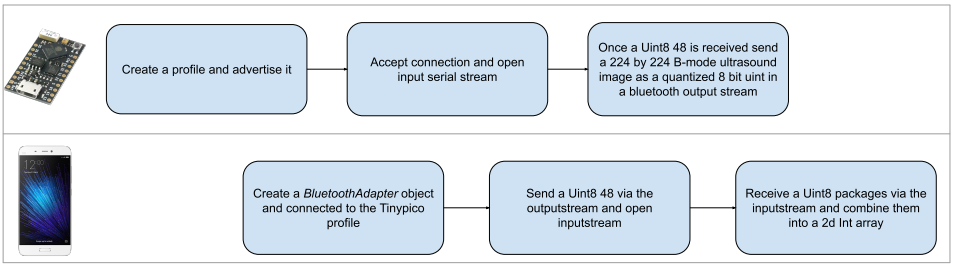


Figure 4.3: Actions taken by both the Android phone and the tinypico in order to send a ultrasound B-mode image via Bluetooth from the tinypico towards the Android phone.

4.3.2. TEMPLATE MATCHING

To implement Template matching within an android environment the OpenCV Module was used. This module provides pre-made function that uses the android phones GPU if available to implement Cross-correlated template matching. The code used in the implementation is described in [52].

4.3.3. VIVO LAS VAGUS

To implement VLV within an android environment the usage of Tensorflow LITE is recommended as android has the *AutoML* feature to automatically integrate a Tensorflow Lite model within an android environment optimized for usage of GPU's. To convert the to tinyML a conversion scheme shown in fig 4.4 is implemented and is realized in [52]. When the generated Tensorflow LITE model is imported within an android environment AutoML automatically creates a class out of it and shows a description on how to implement it. This guide can simply be followed requiring the user only to recreate the

postprocessing steps taken in the original python version. Within [52] the realised code is shown.

Figure 4.4: Flowchart for conversions of original PyTorch model of VLV towards the AutoML accepted Tensorflow Lite model.

4

4.4. IMPLEMENTATION OF A TEMPLATE MATCHING ACCELERATOR WITHIN AN FPGA

Within this thesis a template matching accelerator has been created to be implemented within the Ultra96V2, this was done using verilog code with the full code detailed in [52]. Within the following paragraphs an overview is given of all the functionality and considerations needed to implement template matching within a FPGA.

4.4.1. DESIGN CONSIDERATIONS

The aim of the Template matching on an FPGA is to create an accelerator which applies a cross correlation algorithm as detailed in formula 4.6 into a HDL representation.

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{4.6}$$

To perform this calculation within a FPGA accelerator environment, a design was created using a computation unit that calculates a single value of a pixel. This computation unit is controlled via an external controller, such as a microcontroller. This external controller controls which templates get cross correlated with which cut outs of the target image.

This design has a major advantage that it easily allows the user to add parallelism to the overall design by increasing the amount of computation units. The number of computation units directly corresponds to the amount of time needed to calculate, e.g. upgrading from 1 computation unit to two will cut the overall time needed to complete cross correlation over the targeted image by half, as such the amount of resources used for adding another computation unit also increase almost linearly.

The overall design was created with flexibility in mind, as such a streaming input has been chosen as a way for the master to upload the template and the cropped target image, allowing a wide range of masters to be implemented.

4.4.2. IMPLEMENTATION

To create a computation unit running a normalized cross correlation template matching algorithm as described in formula 4.6, the following IP's are created:

1. Nominator
2. int_to_fractional
3. denomsquare
4. divider

The connections between these ip's are shown in Figure 4.5.

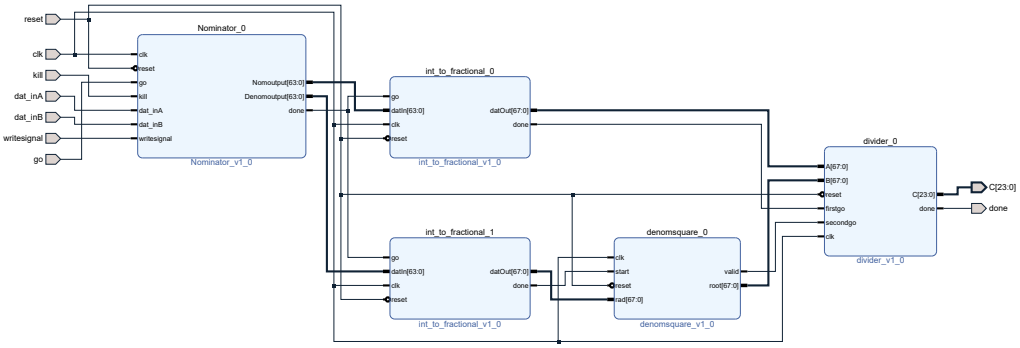


Figure 4.5: A diagram showing all the IP's used with the template matching computation unit created within this thesis and the connections between all of the IP's.

Looking at the general design it can be simplified as a Blackbox diagram shown in Figure 4.6. The overall design expects a 30 by 30 original greyscale image linearised as an 8 bit uint 1 dimensional array ranging from the top left to the bottom right and a 30 by 30 template greyscale image linearised as a 8 bit uint 1-dimensional array ranging from the top left to the bottom right. These inputs are written simultaneously to memory via a streaming interface activated when the write signal receives a high input which after one clock signal immediately gets lowered to a low input. After 7200 clock cycles the input data and the template data is written.

With the Memory block filled, the *go* input is ready to receive a high input. Once this is done after 7105 cycles, or after the done output returns a high signal, a 24 bit fixed point number, with the first 10 bits representing the integer part and the 14 bits to the left representing the number after the dot is outputted. This 24 bit number represents the Cross-correlation factor and can be seen as the main output of the computation unit. The memory can be reset to an empty state and the calculation process can be stopped via setting the *reset* signal high, alternatively the calculation process can be stopped without emptying the memory blocks via setting the *kill* input signal high.
Within [52] the code for the IP's which are described within the following paragraphs in detail.

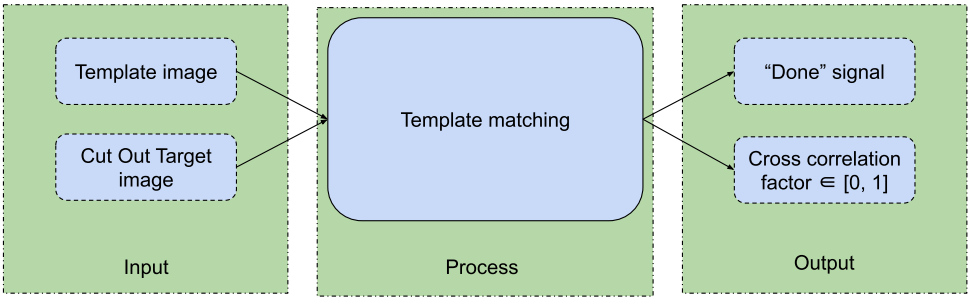


Figure 4.6: A black box diagram showing the relation between the input and the output of the Template matching computation unit, do note that the Cross correlation factor is represented with a 24 bit fixed point number.

4.4.3. LOADING AND CALCULATING THE SET-UP VALUES OF THE DENOMINATOR/NOMINATOR

Nominator is a IP created within this thesis containing a state Machine comprising the following states: idle, calculate, write, and finish.
Outside of this state machine the reset and the kill command exist.
The Reset acts as an initializer for the Flip flop busses and the LUT busses used.
The standard State of the state machine is the Idle state, the only functionality this state has, is to check the *go* input and the *writesignal*, once one of these values goes high the state changes into *calculate* and *write* respectively.
A state diagram is shown in Figure 4.7

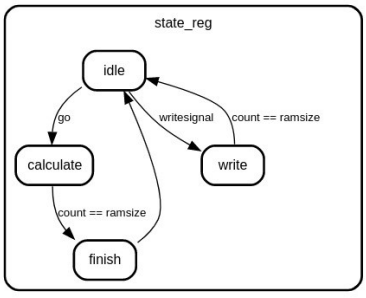


Figure 4.7: State diagram of the Nominator IP, note that the *finish* state remains in its own state for only one cycle after which it automatically reverts back to the idle state.

Once the write state is reached it remains at this state until the write is complete or the reset or kill is high. The functionality of the write state is described in the flowchart of Figure 4.8. Do note that the 2D array A and B is a LUT Array within the implementation within the FPGA.

The calculate state is reached it remains at this state until the write is complete or the reset or kill is high. The functionality of the write state is described in the flowchart of

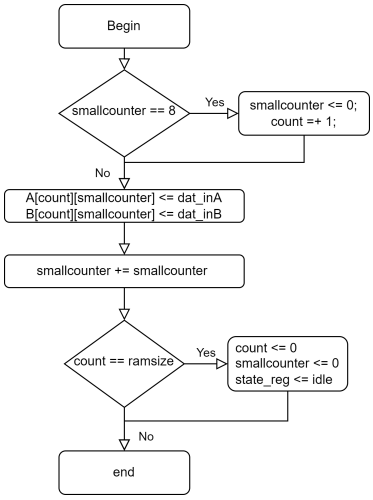


Figure 4.8: A flowchart depicting the information flow of the *write* state in which via a streaming inputs dat_inA and dat_inB two LUT registers get filled, do note that this begin and end occurs within the frame of one clock cycle.

Figure 4.9. The multiplication of the LUT elements A and B is implemented within the FPGA environment as two flip flop arrays using arithmetic units within one clockcycle.

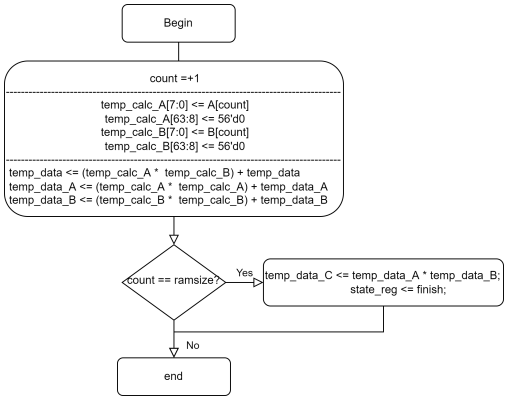


Figure 4.9: A flowchart depicting the information flow of the *calculate* state which loop through all the LUTS and calculates the base nominator and denominator values used to calculate the normalized cross correlation factor, do note that this begin and end occurs within the frame of one clock cycle.

The finish state can be seen as soft reset in which all of the used LUTS and Flip-Flops are reset to their original state and sends a high signal through the *done* ouput signaling the connected IP block to start running. After one clock cycle the state is reset to an idle state.
Once the finish state has been reached the integer output needs to be converted into a

fixed point representation for accurate division and square root by IP's further down the pipeline, this is done via the 'int_to_fractional' IP. The dataflow of the int_to_fractional IP is described in figure 4.10. The main purpose of this IP is to convert the input 57 bit integer into a 67 bit Fixed-Point.

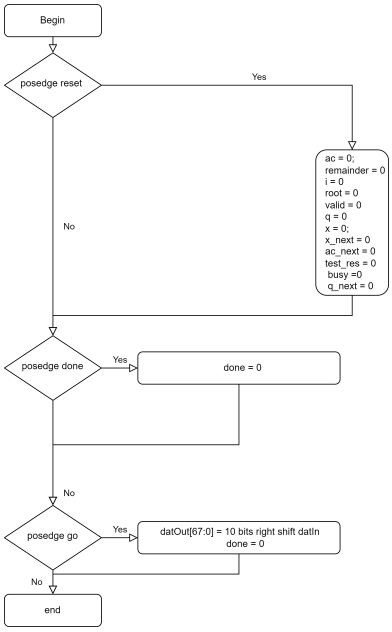


Figure 4.10: flow chart of the *Int_to_fractional* IP, do note that this begin and end occurs within the frame of one clock cycle.

4.4.4. IMPLEMENTATION OF SQUARE ROOT OF THE DENOMINATOR

Denomsquare is a IP block that convert a 68 bit fixed point with 10 fractional radicand bits into a 68 bit fixed point with 10 fractional square root. Within this Verilog implementation the following square root algorithm [53] is taken a converted into a Verilog representation.
The full flowchart of this IP can be seen in appendix E, but for ease of overview the following flowchart is presented with pseudocode to represent how in all its bare basics the square root is calculated.
For this basic implementation 4 registers are generated.

1. X - input radicand
2. A –register holding the current value being worked on
3. T - result of sign test
4. Q - the square root

Within Figure 4.11 the flowchart is shown for this bare implementation. A constraint of this implementation is that the input radicand will always have to be fully divisible by 2.

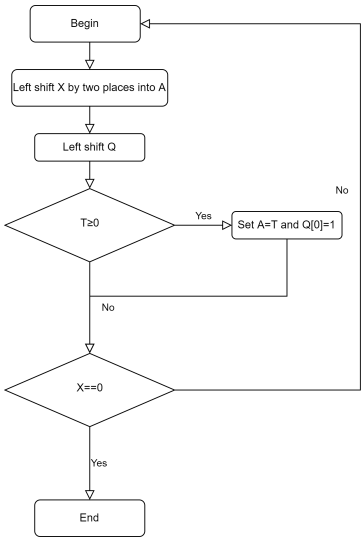


Figure 4.11: Simplified flowchart depicting square division from to input radicand X to the output square root Q„ do note that this begin and end occurs within the frame of one clock cycle.

4.4.5. IMPLEMENTATION OF DIVISION

Divider is a IP block that convert a 68 bit fixed point with 10 fractional radicand bits into a 24 bit fixed point with 10 fractional. Within this Verilog implementation the following long division algorithm [54] is taken a converted into a Verilog representation.
The full flowchart of this IP can be seen in appendix F, but for ease of overview the following flowchart is presented with pseudocode to represent how in all its bare basics the square root is calculated.
For this basic implementation 4 registers are generated.

X - input dividend
Y- input divider
A –register holding the current value being worked on
Q - the quotient

Within Figure 4.12 the flowchart is shown for this bare implementation.

4.5. SYSTOLIC ARRAY ACCELERATOR IMPLEMENTATION VIVO LAS VAGUS WITHIN AN FPGA VIA NNGEN

Within this paragraph the VLV implementation on the FPGA through NNgen is discussed. NNgen is a framework to convert a software representation of a neural network into a

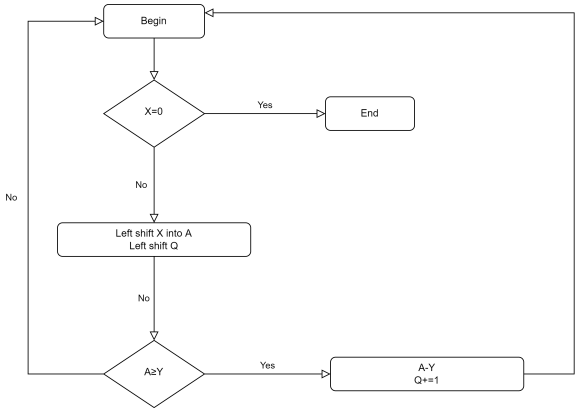


Figure 4.12: Simplified flowchart depicting division from to input dividend X and input divider Y, to the output quotient Q,, do note that this begin and end occurs within the frame of one clock cycle.

hardware representation inside of an FPGA.[55]

4.5.1. ARCHITECTURE

NNgen is a framework to generate FPGA accelerators for ANNs consisting of the basic layers of a neural network, such as convolution, pooling, and full-connection. Veriloggen is used to generate processing hardware circuits in the form of an IP containing Verilog code.
In NNgen, there are two ways to represent a model of a neural network. One is that the programmer explicitly builds a computational graph by combining NNgen operators that support various neural network operations.
The other is to convert trained models built with common neural network frameworks such as PyTorch and Tensorflow into a common neural network format called ONNX (Open Neural Network Exchange). The ONNX model is used as a basis to create a NNgen model, this NNgen model represents both the operators from the original ONNX model as well as Hardware attributes such as right shift and parallelism.
This NNgen Model is converted via Veriloggen to create an IP as discussed before.
The full design flow can be seen in Figure 4.13

The IP that NNgen finally produces looks like the diagram shown in Figure 4.14. The IP is split up in the three operator circuits: The Ram Pool, Containing the BRAM. The Computing Unit Pool containing functionality corresponding to neural network layer eg. conv2d, matmul, and max_pool. Substream Pool contains the arithmetic units. These three operator circuits are connected via custom in-chip networks eg. Substream Interconnect, Memory Interconnect, the Main Thread, and the Direct Memory Access (DMA) controller. The Main Thread acts as a control unit and the DMA controller that transfers data to the outside of the NNgen hardware.

The NNgen IP is a systolic array implementation which causes only one operator cir-

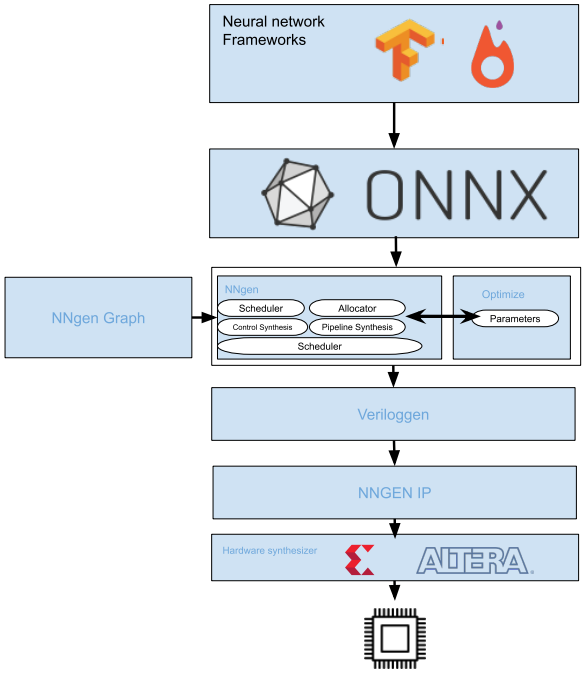


Figure 4.13: Figure depicting the full design flow of NNgen.

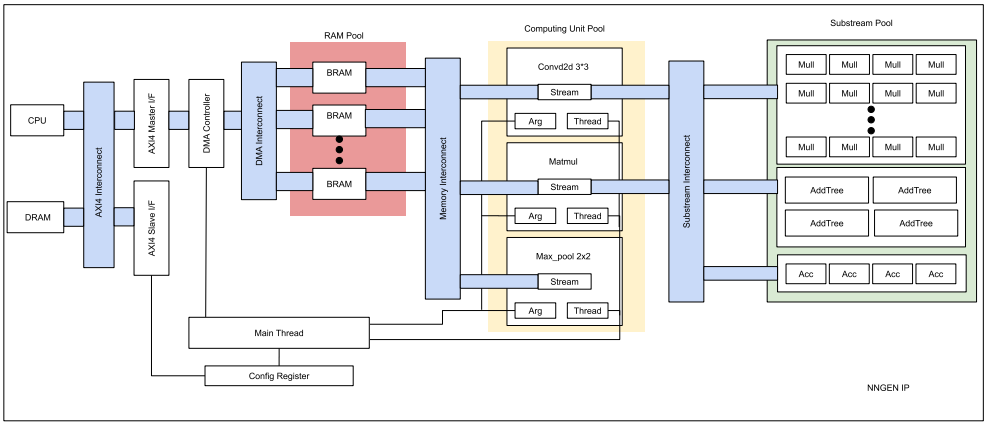


Figure 4.14: Figure depicting the full hardware design of an example generated NNgen IP.

cuit to be generated corresponding to an operator used within the graph even when that operator is used multiple times within the calculation graph. In the RCNN, conv2d and matmul are used 7 to 9 times correspondingly, but only one circuit is generated for each. NNgen is characterized by analyzing computational graphs and generating hardware with the greatest common divisor.

In addition, the arithmetic units used by each operator are collectively generated in the Substream Pool, and each operator shares the pooled arithmetic units via the in-chip network (Substream Interconnect). For example, both conv2d and matmul perform multiply-accumulate operations, but they do not generate their own multiply-accumulate units, but share the same group of units. By sharing the operator circuit and the arithmetic unit, it is possible to suppress the increase in the circuit scale even if the number of layers of the neural network increases. Leading to low amount of resource usage even when large neural network are implemented, this comes at a counter cost of higher latency and higher energy usage compared to a streaming implementation.

4.5.2. DESIGN SPACE EXPLORATION

As the generated NNgen accelerator is an systolic array implemention , every layer and every arithmetic unit is reused within the architecture, as such the flexibility of design space is more limited compared to say, a streaming data flow accelerator.

As the main objective is to simply implement the VLV floating point representation within an HDL representation to be implemented within the Ultra96V2 the following sub-objectives can be added.

1. NNgen implementations of VLV accuracy should be as close as possible to the baseline VLV implementation
2. Power usage per image inference should be as low as possible
3. Inference Time should be as low as possible
4. Resource usage should be as low as possible

Within NNgens design space there are two major variables that influence the aforementioned sub-objectives: The bit width of the quantized weights, biases, input and output and the parallelism, both kernel as well as pixel parallelism. NNgen provides the user with a pre-made quantizer to change the floating point representation of parameters within the original ONNX model to a quantized version used within the generated NNgen Accelerator, however due to lacking results within this thesis an expansion upon this quantizer is suggested.

The hardware parallelism generated by NNgen can be specified in the form of a parameter, separate from the model definition. In this example, the degree of parallelism is specified for each of the conv2d operator (including the matmul operator), pool operator, and element-by-element operation provided via the *par_ich*, *par_och*, *par_col*, and *par_row* parameter.

Due to systolic nature of the implementation every arithmetic unit is reused within a layer, as such there is no clear cut way to measure BRAM efficiency as such the only conclusion that is made regarding efficiency is the one that is made within Appendix B.0.5, namely that pixel parallelism has to increase at the same level as kernel parallelism otherwise resources such as BRAM might not be used optimally, and that an increase of both kernel parallelism as well as pixel parallelism will result in an linear increase of the

latency over all of the created layers, increasing/decreasing the parallelism by a factor of 2 will decrease/increase the latency by a factor of 2 respectively.

Within NNgen the option is given for the user to quantize input/output, weights, biases, scales in steps of 8 bits up 32 bits of bit width using the parameters *act_dtype*, *weight_dtype*, *bias_dtype*, and *scale_dtype* . As the bit width directly influences the resource usage it is necessary to keep the bitwidth of all variables as low as possible to make sure that the resulting neural network is implementable as well as power efficient. NNgen provides the user with a pre-made quantizer to change the floating point representation of parameters within the original ONNX model to a quantized version used within the generated NNgen Accelerator, however due to lacking results an alternative is suggested within this thesis using the *cshamt_out* which controls the amount of right shift per layer. Using this right shift operator and the fact that the generated NNgen computation graphs can be run as Python software by passing input data as arguments a flowchart can be created as shown in figure 4.15.

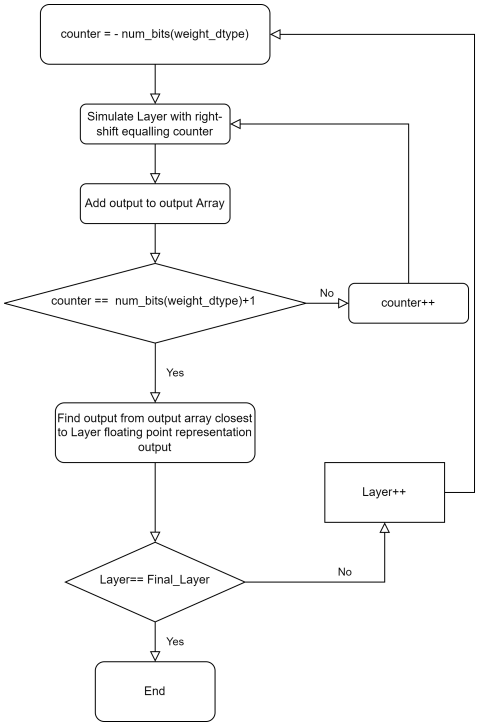


Figure 4.15: A flowchart depicting the workflow a user needs to take to create an accurate quantized model compared to the original floating point model.

Inside, it consists of functions corresponding to each operator that generate the same integer operation results as the hardware. The user can compare this execu- tion result

with the calculation result with the original floating point number to check whether it works as expected before hardware conversion. By passing the input data as an argument, the calculation graph in NNgen format can obtain the same result as the calculation result on the quantisation hardware by soft- ware execution. This makes it possible to check how much the behaviour until final recognition changes due to quantisation before hardware conversion. Within the NNgen software implementation it is possible to manually generate a layer by layer simulation by creating a custom graph in which every layer has its own independent output. if any layers output does not match the original ONNX/PyTorch layer, the amount of right shift should be adjusted within that layer.

4.5.3. WORKFLOW

NNgen's workflow is described within Figure 4.16 Within the following paragraphs the steps are detailed. The PyTorch code broken up into the steps described in figure 4.16 is found in appendix C. As none all of these steps are implemented within the examples provided within the NNgen for the purpose of recreation and future the steps are described with more detailed explanations within appendix C and the code implementing these steps are shown within [52].

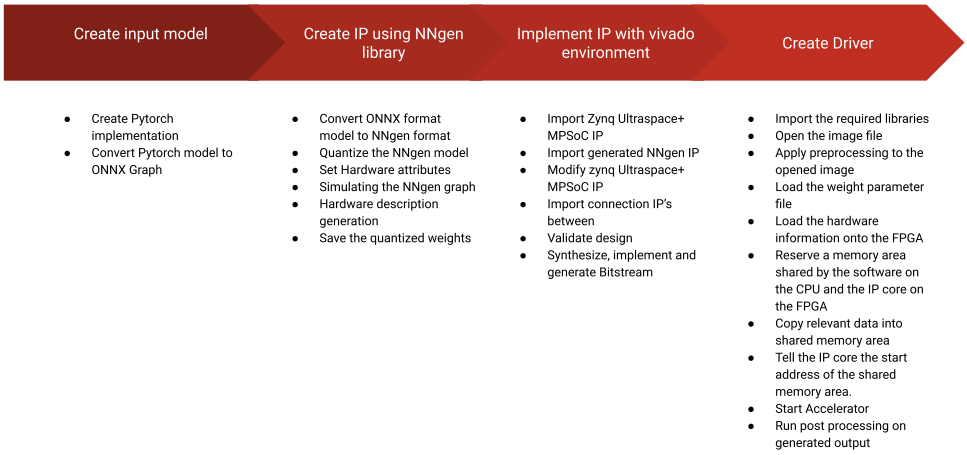


Figure 4.16: A depiction of the workflow within NNgen.

4.6. STREAMING DATAFLOW ACCELERATOR IMPLEMENTATION VIVO LAS VAGUS WITHIN AN FPGA VIA FINN

FINN is a project that aims to provide tools and libraries for creating high-inference, low-latency DNN computation architectures. FINN offers an end-to-end flow a high-level neural network implementation in PyTorch to specialized hardware architectures using an FPGA as show in Figure 4.17. This end-to-end flow starts with the implementation of a NN in Brevitas, which is a PyTorch library for quantisation-aware training and ends

with the generation of a bitstream that can be used to program an FPGA. FINN supports custom architectures, custom precision, and is open source to allow for transparency and flexibility for end-user applications.

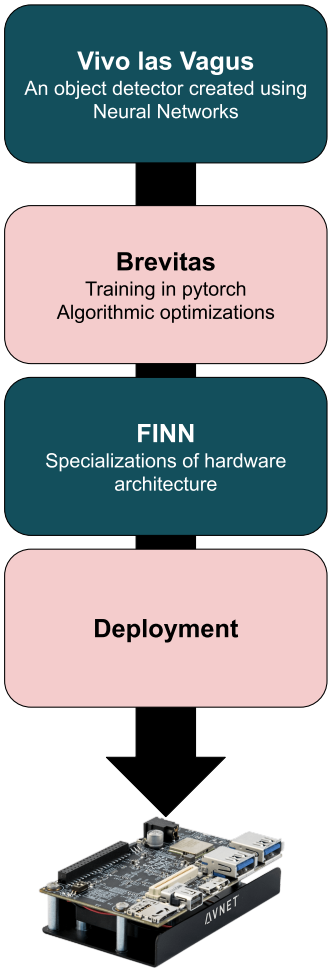


Figure 4.17: Figure depicting the full design flow of FINN.

The neural network accelerators generated by FINN do not use the systolic array architecture, like NNgen's accelerators. Instead, it falls under custom streaming dataflow architectures (DF). The main difference between systolic array architecture and DF architecture lies in the fact that the former executes the network layer-by-layer and uses a more generic hardware implementation to suit multiple layers. DF-style accelerators

have optimized datapaths and the layers can be executed in parallel. They can theoretically be more efficient and provide a lower latency than systolic array accelerators. However, they also have high memory and resource requirements as every layer is implemented individually resulting in duplicate arithmetic units and layers as shown in Figure 4.18, practically this means that NNs implemented as an accelerator within an FPGA using FINN work best if they are small to medium sized. This DF-style architecture results in an extremely flexible implementation in which every layer can be tailored to the requirements of the user.

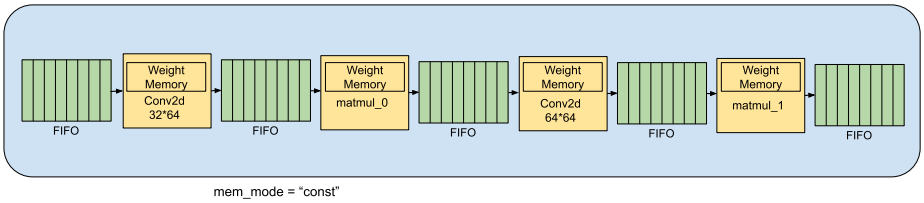


Figure 4.18: Figure depicting the full hardware design of an example generated FINN IP.

The VLV network is small enough to be implemented on an Ultra96V2 board. Its low-latency and low power consumption should make it a good fit for goals set within this thesis.

4.6.1. DESIGN SPACE EXPLORATION

As previously mentioned FINN allows the user to customize every layer within the NN accelerator to its own requirements such as inference time, power and resource usage and accuracy.

As the main objective is to simply implement the VLV floating point representation within an HDL representation to be implemented within the Ultra96V2 the following sub-objectives can be added.

1. FINN implementations of VLV accuracy should be as close as possible to the baseline FINN implementation
2. Power usage per image inference should be as low as possible
3. Inference Time should be as low as possible
4. Resource usage should be as low as possible

Within FINNS design space all of the sub objectives can be interfered using the DF architecture as every layer can adjust its levels of bit width, parallelism, type of data storage, and types of primitives used in the data storage. This is done via the Brevitas implementation and the aforementioned transforms inherent to FINN. Within these next

paragraphs the solutions created to get optimal results for the sub-objectives are detailed.

The typical start of the conversion process is to create a neural network description in PyTorch and train it with Brevitas. Brevitas is a PyTorch library for quantisation-aware training and allows for exporting models suited for the FINN compiler flow; models are exported in ONNX format with datatype annotations to weights to enable quantizing the weights to datatypes smaller than 8 bit integers.

Brevitas implements a set of building blocks at different levels of abstraction to model a reduced precision hardware data path at training time, additionally, a super-set of quantisation schemes implemented across various frameworks and compilers under a single unified API is supported.

The main objective of the Brevitas PyTorch implementation is to create a quantized representation of the VLV network that will result in an as resource light as possible HDL representation on the FPGA. The resource usage directly correlates to the bit width used within the individual layers as the target device is relatively small the average bit width over the entire model should be kept as low as possible.

This can be done manually as the bit width in a layer can be manually set via the parameter *weight_bit_width* and *act_bit_width*, note that the default bit width is set to 8 bit. The model can simply be trained in a similar vein as the original PyTorch implementation. To decrease the resource usage the user is expected to manually lower the bit width parameter values for each layer per run while making sure that the overall accuracy of the NN does not decrease. As such if the objective is to decrease resource usage of the VLV implementation with a bit width range of 2 bits to 8 via decrease of bit width of individual layers the theoretical amount of runs needed to find the optimal application would require $6^{(10)} = 60466176$ runs, where 6 stands for the bit width range and 10 for the number of layers.

As far as the author is aware there is no work yet that implements automatic bitwidth scaling versus the accuracy of the model, as part of this thesis project, a new method using a loss function for Brevitas was developed which automatically tries to decrease the overall bit width of the model without losing any form of accuracy, allowing VLV and by extension, any other NN implemented within Brevitas to have a as time-efficient and minimized user input training as possible, while getting a as accurate and resource effective implementation within FINN. This would also be a significant gain towards the objective of automating the process of creating an object detector within an FPGA as the user will only have to load in the dataset and run the python file without needing any in-depth knowledge of NNs nor of hardware. Finding a balance between making the model more efficient (i.e. smaller bit width) without sacrificing accuracy is a non-trivial problem. It can be looked at as a multi-objective optimization problem or as a constrained optimization problem.

The various ways of implementing the effects of bit width can be quite varied, the following loss function 4.7 has been chosen for its simplicity to implement bit width loss in

tandem with the regular VLV loss function, within this function n equals the batchsize.

$$BitLoss = \frac{\sum_{e=0}^{n} avgbit_e^c}{n} \tag{4.7}$$

With $avgbit$ being defined in formula 4.8.

$$avgbit = \frac{(\sum_{e=0}^{n} Bitwidth(e))}{n} \tag{4.8}$$

With e being a parameter within VLV and n the total amount of adjustable parameters within VLV.
The variable c should be run through a sweep with each training session starting from scratch to retrieve the optimal size of the accelerator while achieving a precision that is similar to the floating-point representation of VLV. The total loss function of the Brevitas implementation of VLV is summarized in function 4.9.

$$\frac{\sum_{e=0}^{n} YoloLoss_e + BitLoss_e}{n} \tag{4.9}$$

The FINN-generated DF architecture includes the implementation of loop parallelism and task parallelism, as well as the ability to set data parallelism. Two parameters are required for FINN-HLS layers representations: PE and SIMD. These parameters specify the number of processing elements (PE) within each compute unit (layer), and the number of Single Instruction, Multiple Data, (SIMD), lanes per PE, do note that this type of parallelism only applies to so called MVAU operator used for convolution layer within FINN, Different types of layers may have different levels or parallelism. These parallelism factors within the FINN are called the folding factors. Only kernel parallelism is available within Convolution layers, however, the experimental branch has pixel parallelism. This is not covered in the thesis.

It is not easy to set the right level of parallelism for each layer. The parallelism factors can be used to adjust two parameters: inference and resource usage. The goal of a parallelism factor is to optimize the inference while preserving the resource limits of the target device. A designer would aim to adjust the parallelism to match the latency of each layer (compute unit).

The goal of this work is to consume as little power as possible and fit within the Ultra96V2 with a 2 second inference. To achieve the inference goals, it is recommended to choose a low clocking speed with a slight degree of parallelism.

Note that the folding factors of a layer have a linear relationship with the latency of that layer, increasing/decreasing the parallelism by a factor of 2 will decrease/increase the latency by a factor of 2 respectively. The logic and utilization of memory resources will be affected if the layer's parallelism is adjusted. The expectation is that increasing the parallelism will result in an increase in the logic resources (i.e. DSPs and LUTs will vary depending on the logic resources used to implement the arithmetic. The shape of the array that holds the weights will be determined by the parallelism factors PE or SIMD.

The total weight array is composed of MW·MH· W_B bits. With W_B bits referring to the number of bits representing a weight value, and MW and MH being related to the number of input and output channels respectively. . Furthermore, note that each PE will have SIMD number of multiplication/additions in parallel between the input pixels and weights
Therefore, when a convolution layer implemented within FINN reads in a new data packet for processing, it should also read PE· SIMD · $W_B$bits from the weight array. Experiments have shown that this results in a weight array of width as expressed via variable W,and depth as expressed via variable D,are expressed by formula 4.10 and formula 4.11 respectively

$$W = PE \cdot SIMD \cdot W_B \tag{4.10}$$

$$D = \frac{MW \cdot MH}{PE \cdot SIMD} \tag{4.11}$$

As certain resources have a fixed shape, e.g. BRAM within the Ultrascale environment can store 36 kilobits; either as a single 36 kbits RAM module or two independent 18 kbits RAM modules with a fixed amount of write ports influencing how arrays are partitioned. For example, the width of the weight array is not utilizing the full width of the hardware memory component, the memory component will not be fully occupied and the remaining bits along the width are wasted. Especially for less flexible memory components, such as BRAM, this might lead to severe underutilisation of the memory component. FINN gives the user the option to perform an analysis which gives a layer by layer output regarding BRAM resource usage, this is used to calculate the BRAM efficiency. The BRAM efficiency is calculated as formula 4.12.

$$BRAMeff = \frac{W}{BRAM16c} \tag{4.12}$$

With $BRAM16c$, the 16 bit bram estimated capacity calculated within formula 4.13.

$$BRAM16c = BRAM16_{est} \cdot 36 \cdot 512 \tag{4.13}$$

With $BRAM16_{est}$being calculated as Formula 4.6.1

$$\text{BRAM16}_{est}(W,D) \triangleq \begin{cases} \lceil \frac{D}{16384} \rceil, & \text{if } W = 1 \\ \lceil \frac{D}{8192} \rceil, & \text{if } W = 2 \\ \lceil \frac{D}{4096} \rceil \lceil \frac{W}{4} \rceil, & \text{if } W \leq 4 \\ \lceil \frac{D}{2048} \rceil \lceil \frac{W}{9} \rceil, & \text{if } W \leq 9 \\ \lceil \frac{D}{1024} \rceil \lceil \frac{W}{18} \rceil, & \text{if } (W \leq 18) \vee (D > 512) \\ \lceil \frac{D}{512} \rceil \lceil \frac{W}{36} \rceil, & \text{otherwise} \end{cases} \tag{4.14}$$

The wish for the BRAM efficiency to at least be higher than 50% for every layer inside of VLV, as this means that the BRAM usage is nonoptimized allowing the degree of parallelization to be doubled for that particular layer without using more BRAM resources.

Within FINN the ability is set for individual layers to use different types of primitives for weight memory. The goal of any FPGA design is to have a balanced design in which

every primitive is utilized to roughly the same degree. Using the *resType* within a FPGA dataflow node attribute primitive types can be selected.
FINN supports two types of the *mem_mode* attribute for the HLS implementation of activation layers. This mode controls how the weight values are accessed during the execution. Currently, two settings for the *mem_mode* are supported in FINN: Const and decoupled.[8]

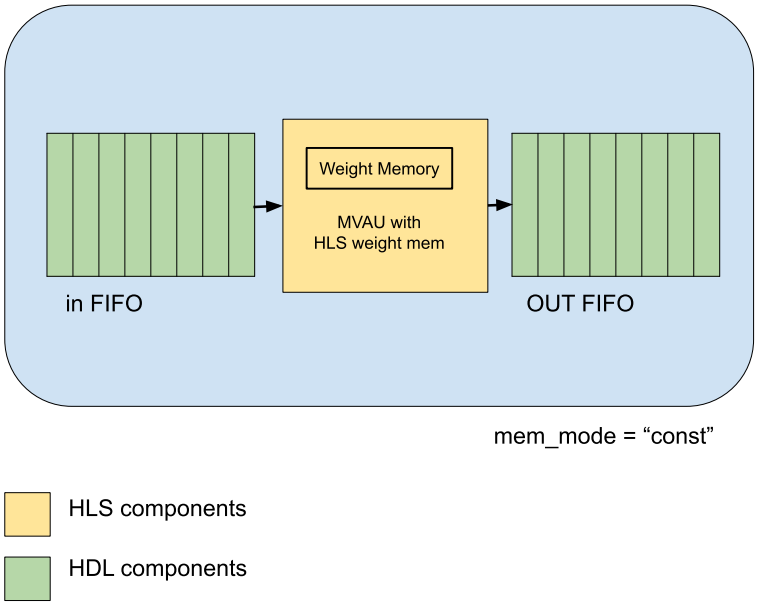


Figure 4.19: Figure depicting the architecture of a MVAU FPGA dataflow Component in *const* mode, note how the weights are baked into to the component. Figure inspired by [8].

The *const* mode has the weights *baked in*. This means that they are part of the HLS code. The weight values are included in the code during the IP block generation and synthesized with it. The user can find the code that generated the resulting IP in the FINN HLS Library. As shown in Figure D.6, the resulting IP block contains an input stream and an output stream, FIFOs are connected to these.
Const mode offers a significant advantage over traditional decoupled modes as they use fewer resources resulting in less power usage. However, because it allows the user to control the weight memory primitives with less precision, resource allocation problems can arise. Vivado HLS may not always produce the correct synthesis.

A different version of the MVAU with three ports is used in *decoupled* mode. The circuit's input and output streams are connected via Verilog FIFOs. A third input is used to stream weights. The user can find the code that generated the IP in the FINN HLS Library. A Verilog weight streamer component retrieves the weight memory from the MVAU and sends

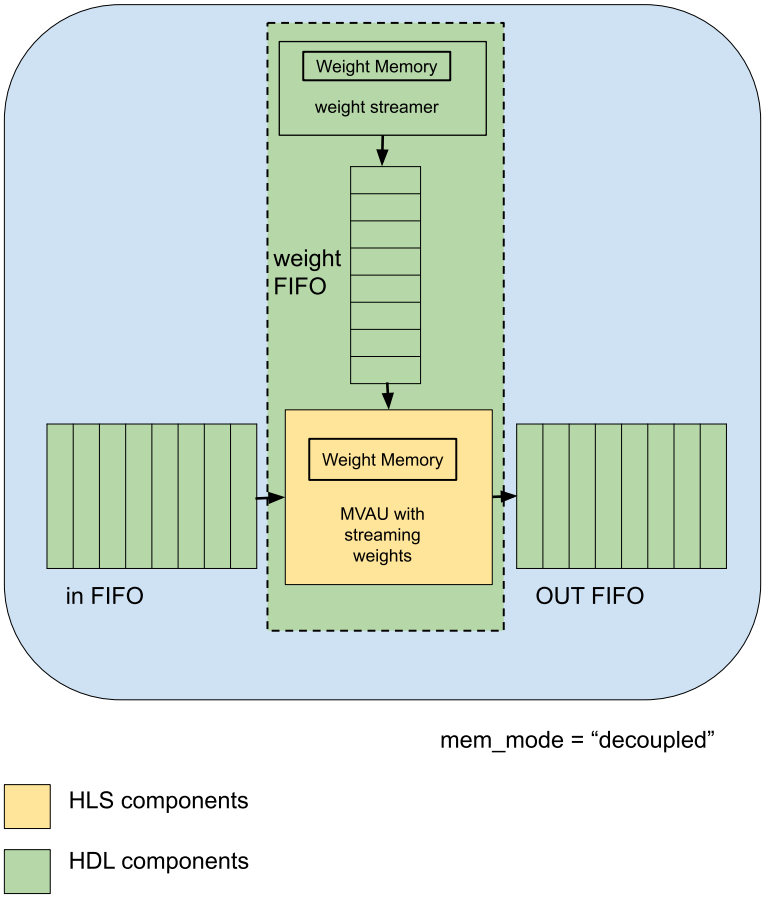


Figure 4.20: Figure depicting the architecture of a MVAU FPGA dataflow Component in *Decoupled* mode, note the added weight streamer with the FIFO attached for loading the weights into the weight memory from a .dat file. Figure inspired by [8].

them the values via the third FIFO. This FIFOis located in FINN. For the IP block generation this component, the IP block resulting from the synthesis of the HLS code of the streaming MVAU and a FIFO for the weight stream are combined in a Verilog wrapper. The weight values are saved as.dat files, and then stored in the weight memory that the weight streamer can read. Externally, the MVAU in decoupled mode provides the same inputs and outputs as the const mode MVAU.

Decoupled mode has a larger resource footprint due to the additional weight FIFO and weight streamer. However, it gives the user more control over which primitive they choose to use. Because it has lower synthesis times than Const mode, and can load different weights through the .dat file, prototyping environments have more functionality using *decoupled* memory mode.

Due to the main objective of the implementation of VLV inside of an FPGA, for the design to be as power and resource efficient as possible as opposed to have a more balanced but heavier resource usage the choice of data type was set at *const*. In an attempt to create a balanced resource usage the final layers of VLV, which are expected to be the largest, have their primitive types set to DSP.

4.6.2. WORKFLOW

The sequence of converting a PyTorch implementation into a neural network accelerator can be broken up into the following parts.

1. Brevitas implementation
2. Tidy-up and define input
3. Streamlining floating-point operations
4. Lowering convolutions
5. Conversion to HLS layers
6. Layer folding
7. Selecting memory mode
8. Set FIFO depths
9. ZYNQ build

Due to the fact that there is no earlier implementation for object detectors within FINN, within this thesis these steps have been implemented for the first time using FINN's available transformations. For the purpose of recreation of this thesis or possible expansion have these transformations been described with more detailed explanations within appendix C and the code using these transforms are shown within [52].

Now that an overview is given of 5 implementations using VLV an overview is given of the results of the efficiency of every implementation detailed within this chapter. This overview is detailed in the next few pages.

5

RESULTS

5.1. BASELINE ACCURACY TESTED ON CPU WITH FLOATING POINTS

To set the baseline standard for all future wearable implementations detailed within this chapter the accuracy of both template matching as well as VLV within a floating point Python environment ran using a CPU, was measured using the accuracy measurement algorithm described within section 3.3.2. This measurement was done over the test set within the overall data split with VLV being trained to 99.4 percent accuracy using the training data set. Template matching is done with only one template used shown in appendix H.

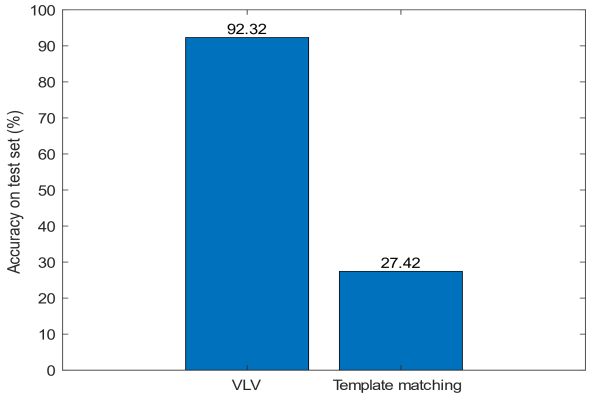


Figure 5.1: A bar chart depicting accuracy of desktop implementation of both VLV as well as template matching.

The VLV implementation is to be expected to be highly accurate while for template matching an extremely low accuracy of around 10 percent is expected based on the previous

works of Videa Pashei et al.[3], as can be seen Figure 5.1 in both VLV as well as template matching the accuracy is relatively high, this can perhaps be contributed towards the static spirit of the created data set. The increase of accuracy of VLV compared towards the template matching implementation does show that NN object detectors are more suited for the task of finding the dynamically shaped vagus nerve.

5.2. RESULTS OF THE MOBILE IMPLEMENTATION

For the mobile implementation a Bluetooth connection between the Android phone HUAWEI Mate 20 and the TinyPico over which a ultrasound B-mode image was send from the test set was sent via the TinyPico over Bluetooth towards the HUAWEI Mate 20 for inference as detailed in section 4.3.

5.2.1. ACCURACY OF THE MOBILE IMPLEMENTATION

With Figure 5.2 the accuracy of the mobile implementation over the test set for both a floating point Tensorflow lite implementation of VLV as well as template matching. As can be seen from Figure 5.2 the loss of data from the Bluetooth transfer is minimal if any exist, nor is there any loss of accuracy from the transformation of the original PyTorch implementation towards the finalTensorflow lite implementation. As expected the template matching implementation behaves the same way as the original Python version on the CPU.

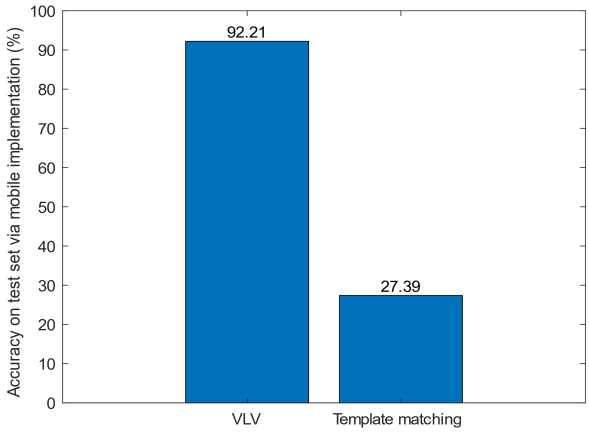


Figure 5.2: Bar chart depicting accuracy of the android implementation of both VLV as well as template matching.

5.2.2. INFERENCE TIME OF THE MOBILE IMPLEMENTATION

The average inference over test set on the mobile phone has been tested on the HUAWEI Mate 20 lite using the Mali-G51 MP4 GPU. The time taken for processing a single image is shown in Figure 5.3 A. To find the true time to process a single image the time taken to

send the target image from the TinyPico needs to be included, since the average time to send a 224 by 224 via Bluetooth is around 6.64 seconds via the implementation written within this thesis a more accurate inference time is shown in Figure 5.3 B.

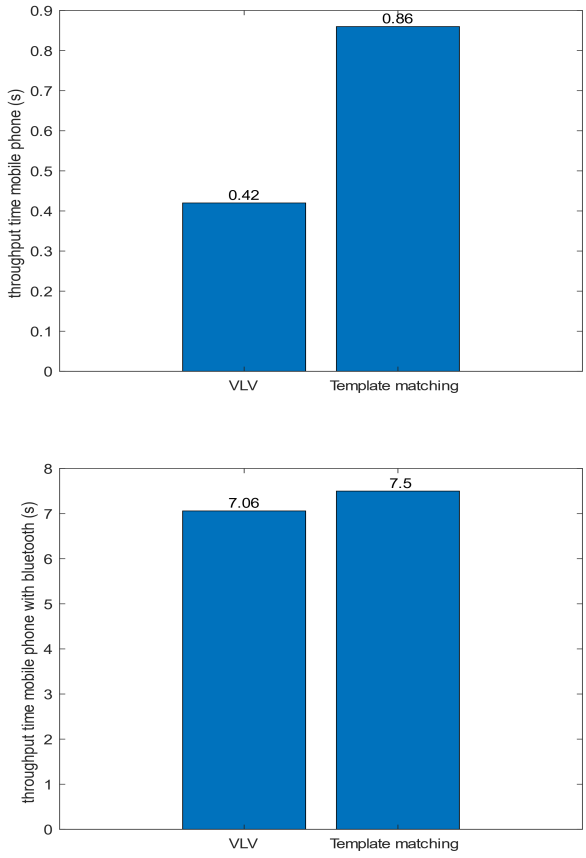


Figure 5.3: Figure A depicts a bar chart of the base average inference time of the HUAWEI Mate 20 lite using the Mali-G51 MP4 of both VLV and template matching with Figure B showing the average inference time including both the base inference of the HUAWEI Mate 20 lite as well as the time taken for the image transfer from the TinyPico towards the the HUAWEI Mate 20 lite.

As shown in 5.3 B this is a significant increase in time taken for inference 5.3 A, while no minimum standard has been set for maximum inference time, these results in the opinion of the author are still unacceptably high.

5.3. RESULTS OF THE FPGA IMPLEMENTATIONS

5.3.1. RESULTS FINN DESIGN SPACE

As FINN allows for a large degree of freedom through its DF style architecture, as such the FINN implementation has a specific section showing results for the two problems detailed within FINNs design space section:

- Under what conditions does the *BitLoss* function for automatic minimisation of bit width over the VLV perform the best?
- At what amount of parallelization is the BRAM efficiency higher than 50 percent over all the layers of VLV?

The parallelism implementation based on the MW and the MH of the generated HLS is written down in Table 5.1 resulting in a BRAM efficiency shown in Figure 5.4. The algorithm used to calculate BRAM efficiency is described in section 4.6.1.

Layers	PE	SIMD	MW	MH	IFMChannels
1	2	3	27	32	3
2	2	8	288	64	32
3	2	8	576	64	64
4	2	8	576	64	64
5	2	8	576	64	64
6	2	8	576	64	64
7	2	8	576	64	64
8	3	8	64	15	64
9	4	15	735	124	1
10	7	4	124	637	1

Table 5.1: Overview of the parallelism and MW and MH as described in section 4.6.1, note that layer 9 and 10 have only 1 channel, with PE parallelizing over the MH dimension while SIMD parallelizes over the MW dimension. As shown the final 2 layers 9 10, are shown to be the largest layers with the highest MW and MH.

As can be seen overall the BRAM is efficiently used however the parallelism could be increased for the initial layer and the second layer resulting in an increase in speed without losing resource usage.
Table 5.2 shows the BRAM efficiency together with the total amount of BRAM and DSP used per layer. Note that this shows that the resource usage is most heavy in the final 2 fully connected/linear layers even when the primitives have been set to DSP within FINN as described in section 4.6.1.
To find the most effective configuration of the VLV, a setup was created to perform a variable sweep. The variable 'c' corresponding to the power of the *BitLoss* function is shown in function 5.1 and further detailed in section 4.6.1. This is swept from 0 to 10 in steps of 1. Within the NN, a seed is set, that the NN starts training without randomisation acting as a nuisance variable. The setup code for this experiment is provided in [52].

$$BitLoss = \frac{\sum_{e=0}^{n} avgbit_e^c}{n} \tag{5.1}$$

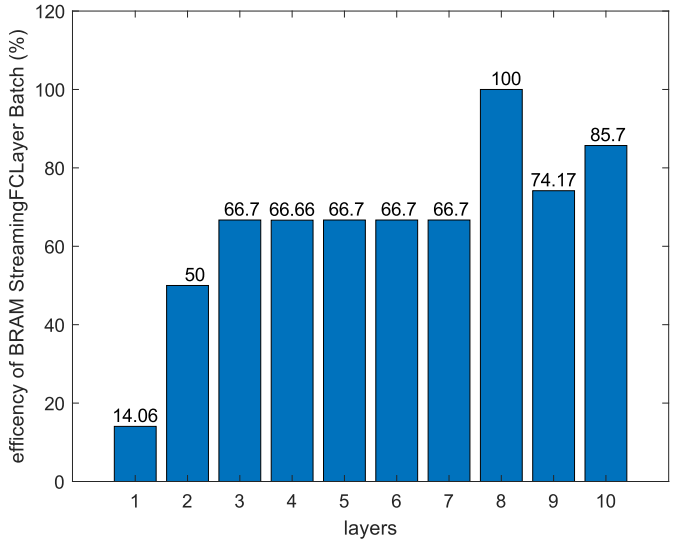


Figure 5.4: A bar chart depicting the BRAM efficiency of the VLV FINN implementation layer per layer, do note that the first two layers have low enough efficiency that parallelism could be increased without resulting in higher resource usage.

Layer	1	2	3	4	5	6	7	8	9	10
BRAM efficiency (%)	14.06	50	66.7	66.7	66	x	66	x	74.17	85.7
BRAM usage	1	6	9	6	6	0	20	0	20	15
DSP usage	0	0	0	0	0	0	0	0	60	28

Table 5.2: This Table gives an overview of the resource usage layer by layer while including the BRAM efficiency estimation based on the formula detailed in section 4.6.1.

Two versions of this experiment have been created, one which initialises VLV over all layers with a bit width of 8, and a version which initialises VLV over all layers with a bit width of 2.

The results of the average bit width across VLV for the variable sweep with an starting bit width of 8 is shown in Figure 5.5A and the corresponding amount of epochs is shown in Figure 5.5B, with the full results shown in appendix I.

As shown in the Figure 5.5A there seems to be a strong correlation between the power of the *BitLoss* function and average bit width until the power variable reaches 6. As can be seen in Figure 5.6A while the bit width for all layers goes down initially it seems to stall at 4 bits idem ditto for power variable set at 7, 8, 9, and 10, there is no concrete answer on why this occurs aside from the *BitLoss* function being seen as too large in comparison

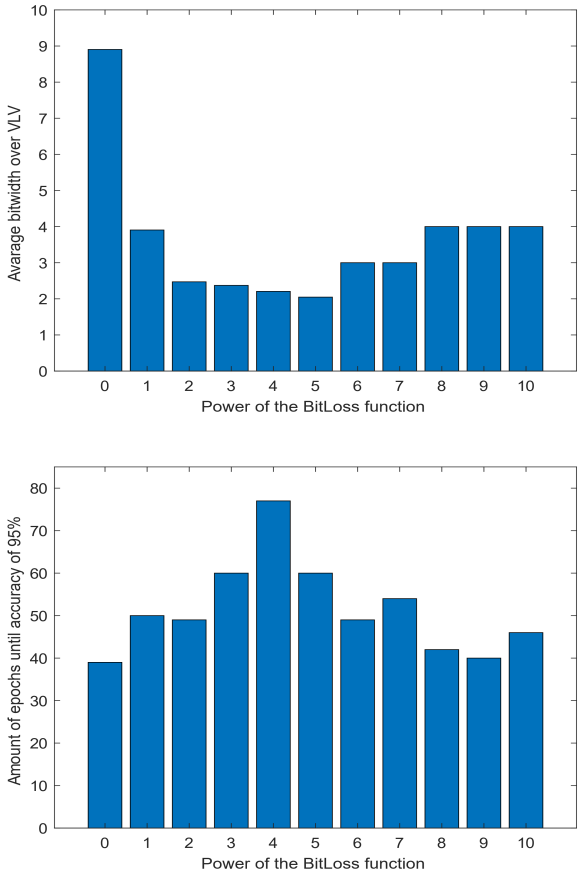


Figure 5.5: Figure A depicts a bar chart of the average bit width of VLV within Brevitas over a power sweep of the variable C depicted section 4.6.1 with the starting values for every layer being 8 bits wide, while Figure B depicts a bar chart of the average amount of epochs required for training over the power sweep of variable C with the starting values for every layer being 8 bits wide.

to the loss from *YOLOLoss* function as seen in Figure 5.6B, however a guess can be made that it relates to the stop condition of a precision of 95%, perhaps training the NN past this stop condition would cause the avarage bit width to go down further.

In contrast the sweep starting at 2 bit shows as strong correlation between accuracy and of power the *BitLoss* function as shown in Figure 5.7A regardless of how large the power of the *BitLoss* function becomes, showing that in all cases the average bit width is lower or equal to its 8 bit width equivalent. What also can be seen is that comparatively the amount of epochs that are needed for the 2 bit implementation increases compared to the 8 bit implementation as shown in Figure 5.7B.

The most efficient average bit width resulting from a trained VLV NN is the one that cor-

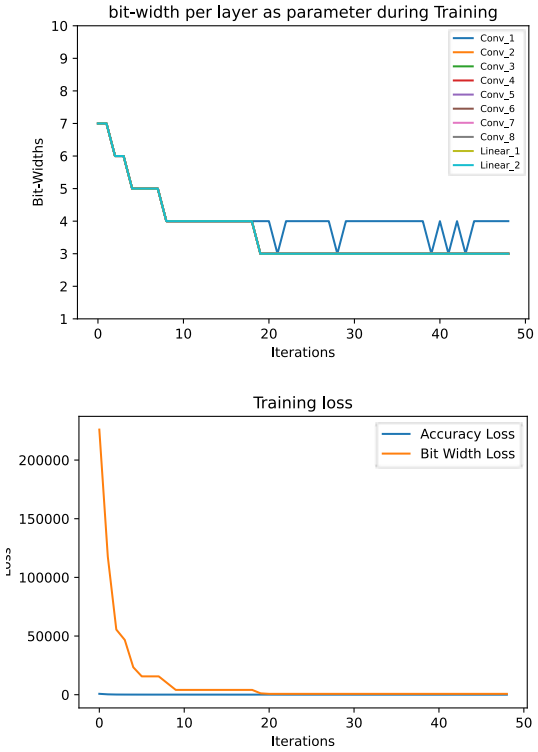


Figure 5.6: Figure A depicts a bar chart of the average bit width of VLV within Brevitas over a power sweep of the variable C depicted in section 4.6.1 with the starting values for every layer being 8 bits wide and the variable C being set a value of 6, note how all of the layers of VLV are equal to the *Conv1*, and *Linear2* layer giving them the appearance of hiding. Figure B depicts a bar chart of the average amount of epochs required for training over the power sweep of variable C with the starting values for every layer being 8 bits wide. Note that Accuracy Loss and YOLO Loss are used interchangeably.

responds to a *BitLoss* function with the power variable set to 10 with the initial bit width set to 2. As can be seen from the Figure 5.7A the bit width over all the layers within the VLV NN remain static throughout all the epochs and barely change, this due to the overbearing presence of the *BitLoss* function compared to the *YOLOLoss* function as shown in Figure 5.7B.

The almost static bit width throughout all layers under a low initial bit width is a recurring pattern, as can be seen in appendix I.

This is in contrast to the implementations in which the initial bit width starts at 8 bits, with a more dynamic bit width throughout all epochs shown in Figure 5.9A, even when the *BitLoss* function lays dominant in later epochs as shown in 5.9B, the act of lowering the bit width within a individual layer within VLV has a such detrimental effect on the accuracy that the *YOLOLoss* function again rises to dominance causing the bit width to

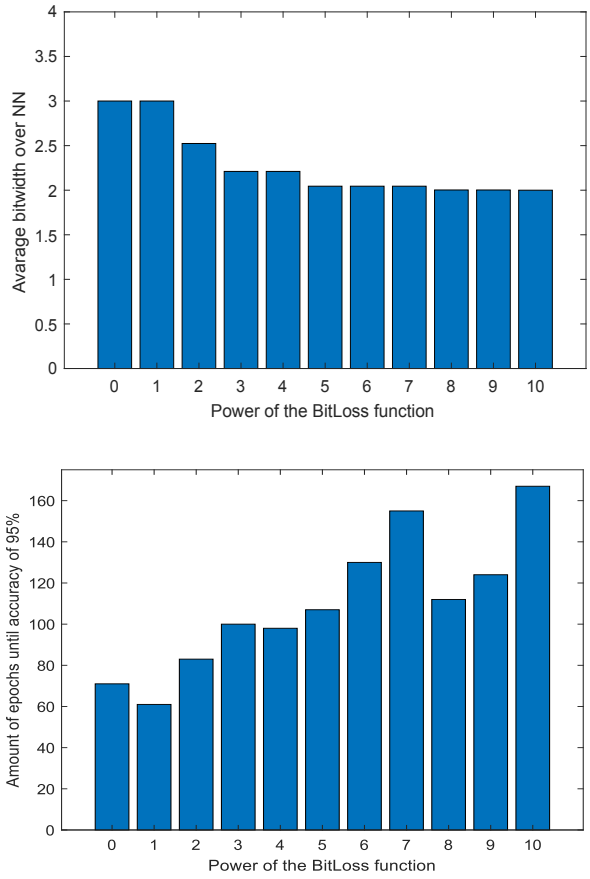


Figure 5.7: Figure A depicts a bar chart of the average bit width of VLV within Brevitas over a power sweep of the variable C depicted section 4.6.1 with the starting values for every layer being 2 bits wide, while Figure B depicts a bar chart of the average amount of epochs required for training over the power sweep of variable C with the starting values for every layer being 2 bits wide.

revert backs to its original state, this could be the major contributing factor as to why starting training from a lower bit width seems to contribute to a higher precision.
For future implementations with new data sets or for a completely different NNs altogether, the best implementation of the *BitLoss* function is to do a parameter sweep in which the power variable is swept from 2 to 10 with the initial bit width set at 2 if the goal is to find the highest accuracy while retaining the lowest bit width via training.

5.3.2. INFERENCE FPGA

The inference time for the FPGA implementations are shown in Figure 5.10, do note that no interface between the template matching implemention detailed in section 4.3.2 and the ZYNQ Ultrascale+ MPSoC, as such the amount of time taken within an implemented

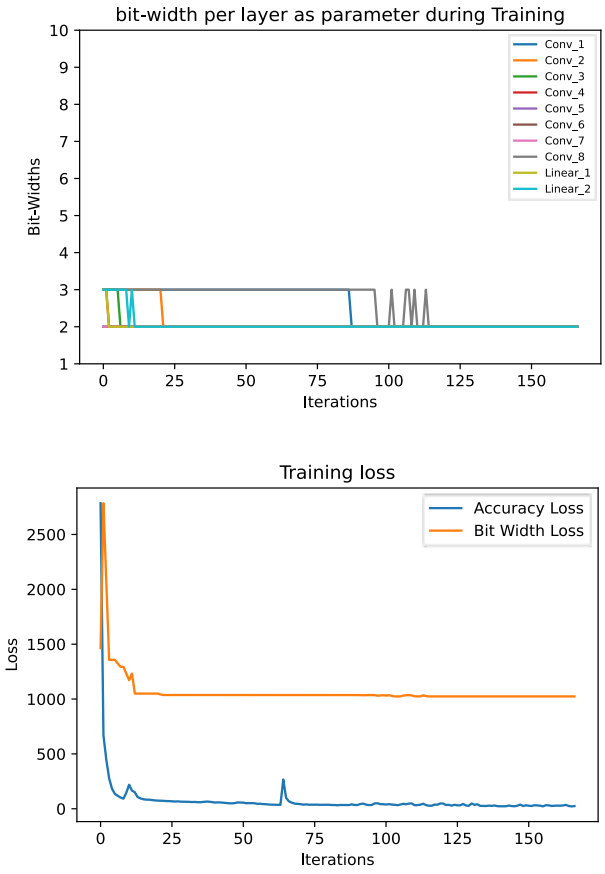


Figure 5.8: Figure A depicts a chart of the individual bit-width of every layer within VLV throughout the training process with the power of the *BitLoss* being 10, note how all of the layers stay mostly static throughout the training, with Figure B depicting a chart of the loss produced by the *BitLoss* aswell as the *YOLOLoss*, note that the *BitLoss* is dominant throughout all the training iterations. Be aware that Accuracy Loss and YOLO Loss are used interchangeably.

simulation is taken as the inference. The test-bench code is shown in [52].

As can be seen the FINN implementation with its high parallelism and streaming dataflow structure results in the fastest inference of all the implementations at 0.174 second per image, secondly comes the systolic array implementation of NNgen without any parallelism build in at 0.54 seconds per image. Finally the template matching implementation with a 0.776 seconds per image is the slowest, do note however that for every pixel calculated within the template matching algorithm almost half of the computation time(14305ns) goes into streaming the input template as well as the cut-out of the target image(7200ns), also of note is that this template matching implementation only has one computation unit, doubling the amount would result in roughly half the inference time.

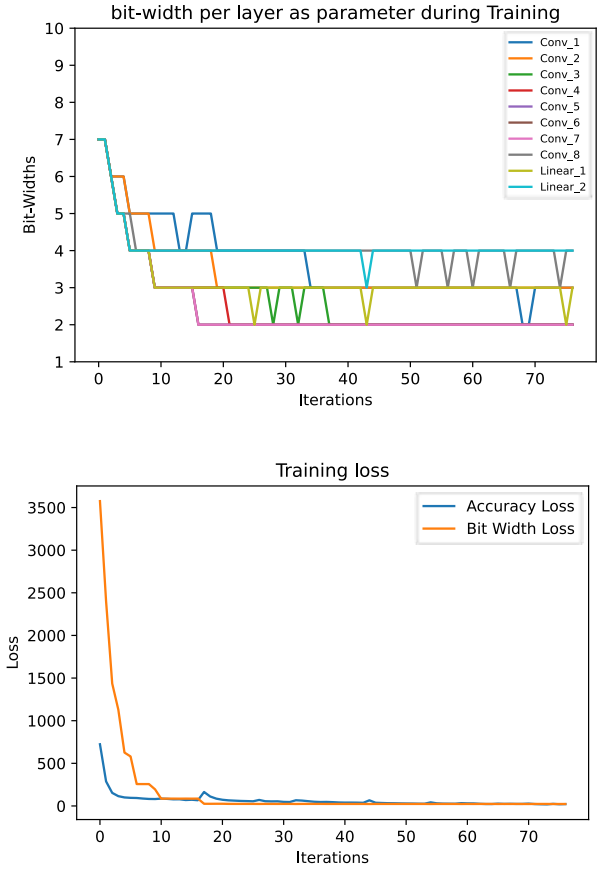


Figure 5.9: Figure A depicts a chart of the individual bit-width of every layer within VLV throughout the training process with the power of the *BitLoss* being 5 and the initial bitwidth set to 8, note how all of the layers stay dynamic throughout the training, with Figure B depicting a chart of the loss produced by the *BitLoss* aswell as the *YOLOLoss*, note that the *BitLoss* majorly dominant throughout the early training iterations while throughout the later training iterations the loss produced by *BitLoss* and *YOLOLoss* is roughly equivalent. Note that Accuracy Loss and YOLO Loss are used interchangeably.

5.3.3. RESOURCE USAGE FPGA

Within the FPGA implementation the following resource usage is defined for VLV implemented within the FINN streaming dataflow architecture with absolute values being showing in Figure 5.11A and percentage based resource usage in the context to the Ultra96V2 inside of 5.11B, the resource usage of NNgen systolic array architecture with absolute values being showing in Figure 5.12A and percentage based inside of 5.12B, and finally the resource absolute resource usage of the template matching computing unit is shown Figure 5.13A, with the percentage based resource usage on Figure 5.13B.

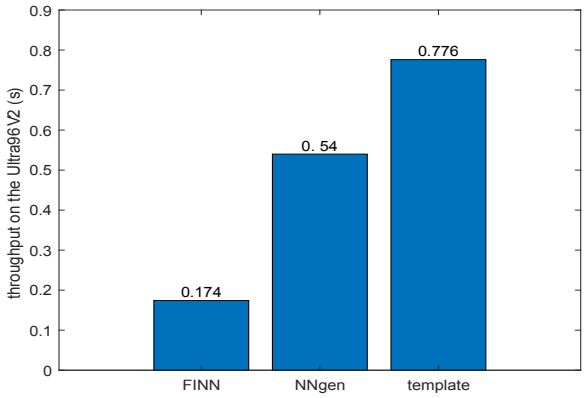


Figure 5.10: A bar chart depicting of the base average inference time in seconds of the VLV accelerators based on FINN and NNgen as well as the template matching accelerator running on the Ultra96V2.

As can be seen from the percentage based resource usage none of the implemented designs have a particularly balanced resource usage, this is perhaps due to the uint8 based quantisation shown in both NNgen implementation as well as the template matching implementation, in which LUTs are used for temporary storage of the template image, and the cut-out of the target image, as well as the multiplication arithmetic units.
The FINN implementation with its modular primitive types has given the most balanced design resulting in the smallest usage difference between every primitive type.
As can be seen in every implementation the resource usage is kept well within the limitations set by the Ultra96V2 as shown in 5.13 B and 3.4, with the only exception being the LUT usage by the template matching set at 55.82 percent and the LUT usage of the FINN VLV Accelerator set 67.03 percent which takes up a significant larger amount of the available 70560 LUT elements, however even though it is an outlier compared to the other primitive usage it still fits comfortably within the Ultra96V2.
The NNgen Implementation of VLV is the only implementation that requires external DRAM, drastically increasing the resource usage shown by Vivado this DRAM estimation is based on the log produced via the NNgen workflow described in section C with the resulting generated log shown in appendix G. As the log shows that 3929791 8 bit addresses are needed for implementation and a single DRAM taking up 32 * 1024 = 32768 bits, or 4096 Bytes per DRAM, 960 DRAM is needed.

5.3.4. ACCURACY FPGA

In Figure 5.14 the accuracy of all 3 FPGA implementations is shown, note that template matching's precision is measured within an implemented simulation.
The quantize aware training from Brevitas used inside of the FINN implementation shows a tremendous increase in accuracy compared to the float-to-int quantisation used within the NNgen implementation. The training set accuracy within the Brevitas model is set to

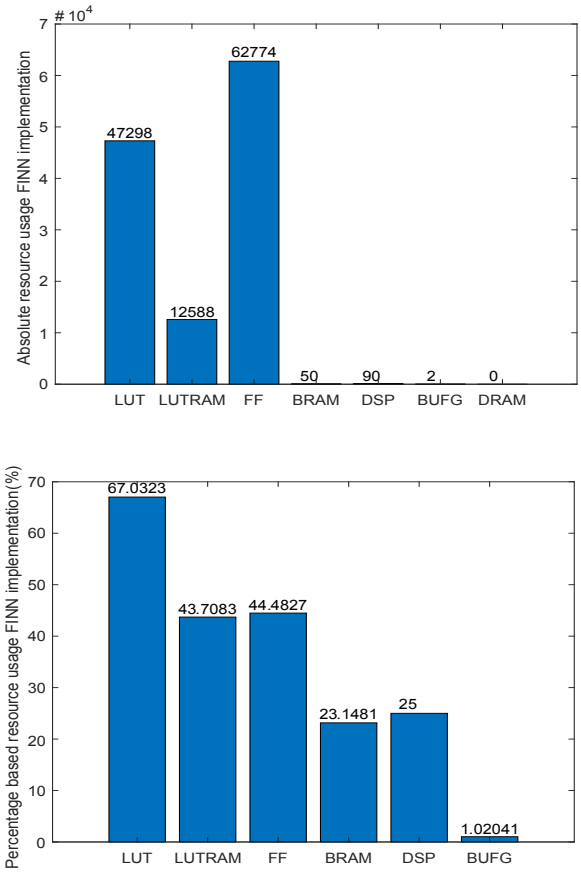


Figure 5.11: Figure A depicts bar chart showing the absolute resource usage of the FINN implementation of VLV while Figure B depicts the percentage based resource usage based on the resources available within the Ultra96V2.

95.6 percent compared to the PyTorch models 99.4 percent accuracy. Template matching's implementation has shown a small difference between the cpu baseline and the simulation but not enough to invalidate the implementation.

5.3.5. POWER USAGE FPGA

Within this chapter two approaches will be taken to measure the power usage of the implemented FPGA designs. Firstly Vivado's automated power usage measurement and a real life test setup using the Ultra96V2.

VIVADO ESTIMATIONS

Within Vivado estimation the following results were returned as shown in Figure 5.15A leading to the following temperature estimations shown Figure 5.15B.

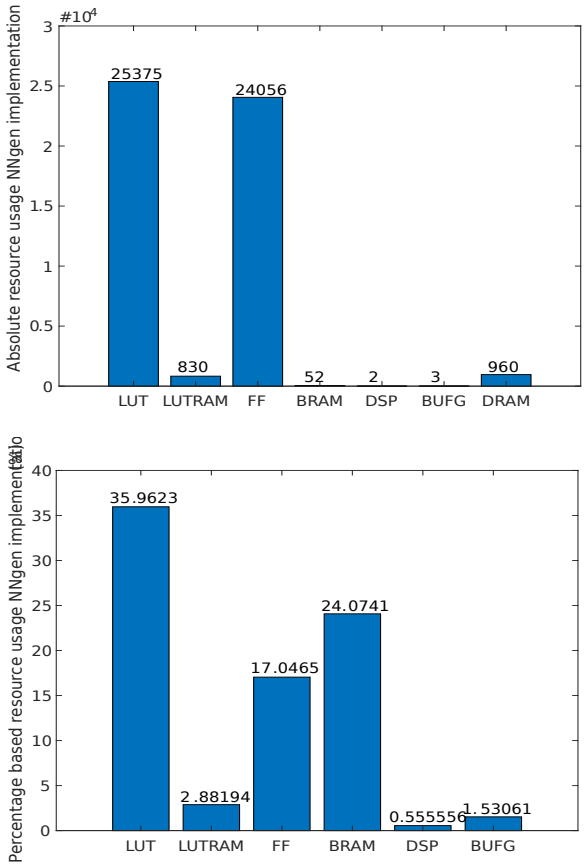


Figure 5.12: Figure A depicts bar chart showing the absolute resource usage of the NNgen implementation of VLV while Figure B depicts the percentage based resource usage based on the resources available within the Ultra96V2, note how there is 960 external DRAM required which is not required within any of the other implementations created within this thesis.

As can be seen there is a large degree of estimated power usage difference between both VLV accelerator implementations compared to the template matching computing unit. This can be explained due to the lack of implementation of a connection between the ZYNQ Ultrascale+ MPSoC IP and the computation unit, resulting in lower but more accurate power usage compared to the inflated numbers found in the both VLV accelerators.

Within all implementations the heat estimation shown in Figure 5.15B while perhaps uncomfortable will not lead to any tissue damage on the wearer allowing a user to safely wear all of the implementations designed.

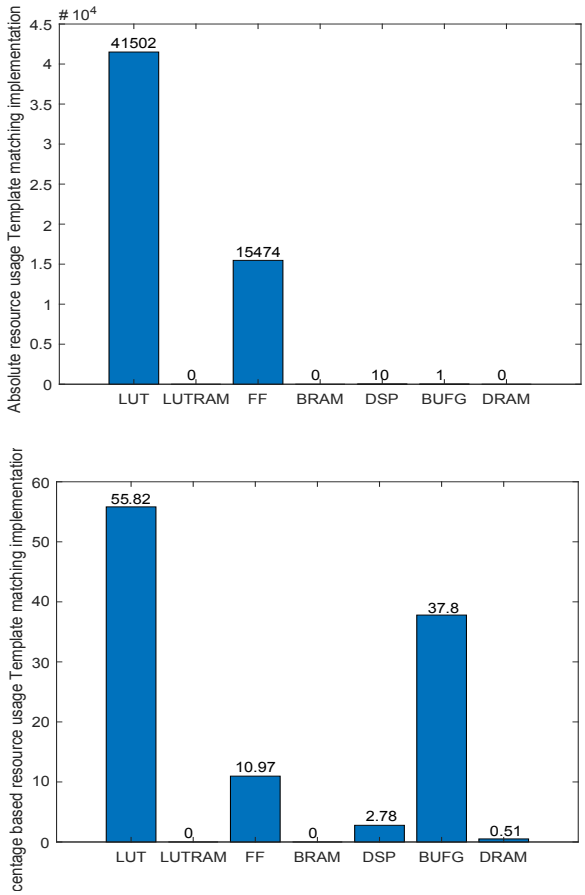


Figure 5.13: Figure A depicts bar chart showing the absolute resource usage of the template matching accelerator while Figure B depicts the percentage based resource usage based on the resources available within the Ultra96V2.

To get a more accurate results the power consumption was measured on the Ultra96V2 board during inference of the trained VLV model in both the NNgen as well as the FINN. The measured voltage of the power supply to the board was multiplied with the measured current to compute the power consumption, with the current being measured via a GW instek GPPP-4323 power supply, with the supply voltage set at 12 volt. The mean power during inference is calculated as a mean over 10 inferences. The power consumption of the different implementations created during this thesis is defined as the difference of the Ultra96v2 board idling and power during inference. The idle power consumption was measured over a five-minute period to be Pidle = 4.14 watt. This results in an average power consumption for both VLV implementations as shown in Figure 5.16. It should be noted however that this form of measurement could be seen as flawed for the NNgen implementation as the external DRAM of the Ultra96V2 is already in use un-

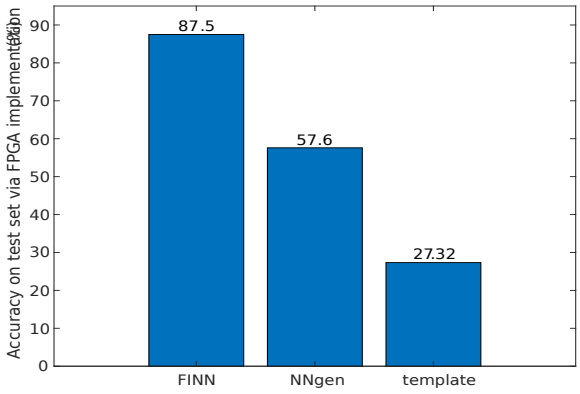


Figure 5.14: A bar chart depicting the accuracy of the VLV accelerators based on FINN and NNgen as well as the template matching accelerator running on the Ultra96V2.

der the shared CMA as described in section C this drastically decreases the actual power usage as this implementation purely measures the FPGA part.
As can be noted the template matching implementation is not measured in real life as no ZYNQ Ultrascale+ MPSoC IP was created.

5.4. COMPARISON TO OTHER WORKS

As finding a nerve via a object detection algorithm implemented within a phone and an FPGA according to the author something that has not yet been done, for the comparison this work will be broken up in two parts: Implementing an object detector for detecting Bio-markers within Ultrasound B-mode images within an CPU, and taking the accuracy as the main comparison point, and Implementing an object detector within an FPGA environment, and taking the power efficiency in FPS/watt as it main comparison point.

5.4.1. COMPARISON OF OBJECT DETECTORS FOR BIO-MARKERS WITHIN AN CPU BASED ON ACCURACY

Aside from a traditional RCNN and template matching implementations created within this thesis a more recent implementation by [11] in which a so called siamese Regional convolutional neural network was created with a attached linear Kalman filter for short term memory object tracking.
As seen in table 5.3 a comparison is made between the works detailed in [11] compared to VLV, within [11] multiple object detetectors created in other works were tested again their Siamese object tracking algorithm upgdSiamFC which is the SIAMFC NN taken from [56] with a linear Kalman filter (LKF) attached, within this work also other alternative object tracking alternatives were implemented and tested, including multiple Siamese neural network architectures but also traditional algorithms such as variations on block matching algorithms. As can be noted the accuracy detection within the work

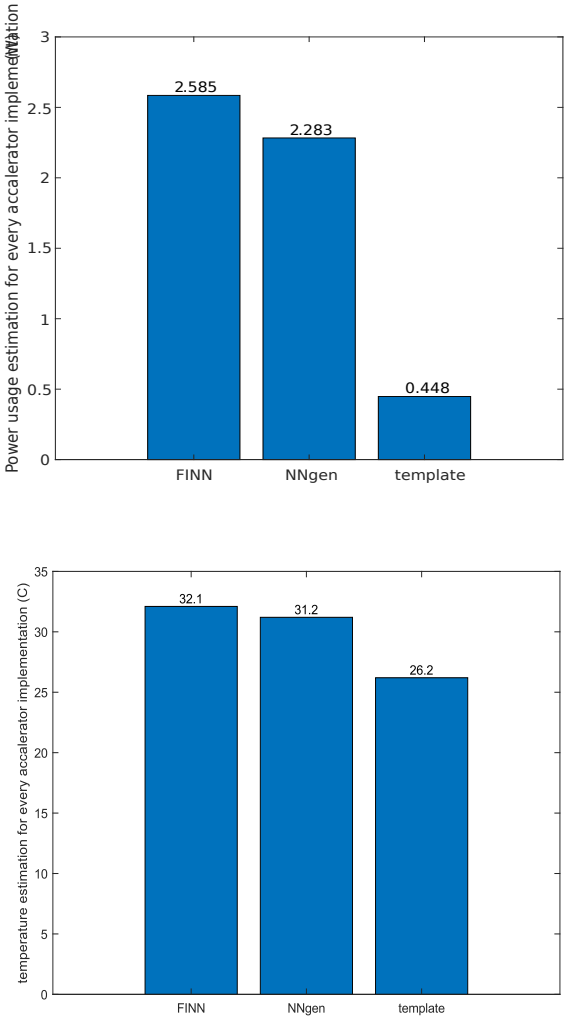


Figure 5.15: Figure A depicts bar chart showing the vivado estimations of power usage in watts of the VLV accelerators based on FINN and NNgen as well as the template matching accelerator running on the Ultra96V2, with Figure B depicting a bar chart showing the vivado estimations of heat generation in degree C.

of [11] is calculated in mm compared to the relative accuracy in percentage within this thesis, this since the shape of the created bounding box is not variable as the SiamFC always results in a bounding box the size of the input template image, another factor is that within the work of [11] the size of the pixels within the B-mode ultrasound relative to real life milimeters is known this is unfortunately case within the test set of this thesis.

As shown in Table 5.3 the training of upgdSiamFC was done using the ILSVRC dataset that consists of camera images of real-world objects, such as animals, vehicles, and

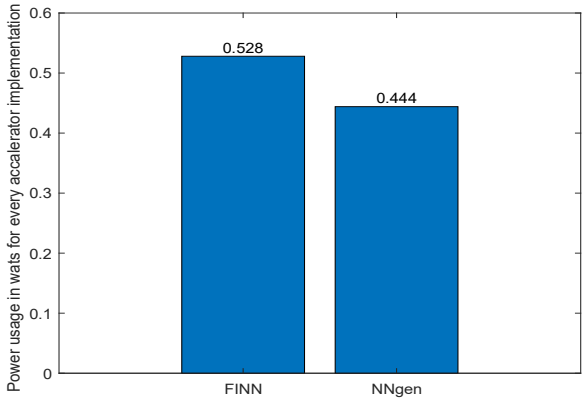


Figure 5.16: A bar chart showing the power usage from real life measurements in watts of the VLV accelerators based on FINN and NNgen within the Ultra96V2.

household items, and the network was not retrained using Ultrasound images for the application. Furthermore, SiamFC does not update the template model or incorporate any motion model.

As can be noted that all works featured in [11] consist of object trackers instead of object detectors, as such they usually depend on the input of the previous frame to detect the object in comparison to a traditional object detector that calculates the location of an object independently of any previous input, object tracking NNs generally have both a higher accuracy as well as a higher inference to object detectors NNs of similar size, however, they require a constant flow of images within every second to return an accurate output. As ultrasound neuromodulation is a process that occurs with delays in between individual modulations to make sure that the targeted nerve doesn't overheat, large moments of inactivity can occur, within these moments of inactivity a traditional object detector can simply not run while an Object tracker is required to run to keep accuracy high, this could have as an effect that object trackers could require more power despite being more power efficient compared to object detectors.

The NN designed in [11] called the upgdSiain has a an overal size of around 5.5 times as large of the VLV base implementation, size here is defined the size floating-point weights and biases of the architecture combined, while no FPGA implementation exist as far as the author knows, an educated guess can be made that the resource usage of upgdSiamFC within a streaming dataflow accelerator would result in a far larger NN unable to fit within the Ultra96V2, as such the only viable implementation within an FPGA would be a systolic array accelerator, however even if it can be implemented, based on the predicted size of upgdSiain the prediction could also be made that VLV will have a faster inference.

The block matching algorithm implementation by [10] has the highest accuracy despite not being a neural network, there is no specific implementation for the algorithm designed within this specific paper, however, there are similar algorithms implemented

Name work	Neural network type	FPS	Size	Accuracy	Device	Input size image	Training Dataset
[11]	Object tracking	4	8.5 MB	1.59 mm	CPU	255x255x3 127x127x3	ILSVRC dataset
VLV	Object detection	5.75	1.9 MB	85%	FPGA	224x224x3	114 US images
[10]	X	X	X	0.72 mm	CPU	X	X

Table 5.3: Table giving overview over the accuracy of a floating point implementation of VLV compared to the works of [11] ,and [10], note how both the accuracy assesment aswell as the test set differ between implementation causing the comparison to be somewhat moot.

within an FPGA showing the possibility of reasonable resource usage for within the Ultra96V2. Within a block matching algorithm a cost function, such as mean absolute difference (MAD), mean squared error (MSE), or Normalized Cross-Correlation (NCC), is used to calculate the similarity between the reference block and the candidate block in an iterative fashion. The predefined search region is searched exhaustively to obtain the best-matching candidate block. Consequently, the cost function is calculated in every iteration, making the method computationally expensive.
All the different works tested within are all variations of the algorithms detailed within this section as such the assumption is made that the predictions made within this chapter hold up for the other works tested within [11]. A important next step for an accuracy comparison is to gather a realistic dataset and have multiple object detection algorithms and run inference over this dataset using the accuracy assessment algorithm discussed within section 3.3.2.

5.4.2. COMPARISON OF OBJECT DETECTORS IMPLEMENTED WITHIN AN FPGA

The most efficient power per frame implementation was chosen, with the FINN implementation with its 0.092 watt per frame or 10.88 FPS/watt was chosen as a comparison to other works, the results are shown in Figure 5.17. For this comparison a small literature search was done using the search term: “object detector fpga power usage” resulting in the following object detectors to be chosen for comparison:Nested RNS [14] Event-Driven OD [13], OD thermal [12], Deep CNN Accelerator [15], and BCNN Accelerator [16].

It should be noted however that measuring power via MPSOC is not a precise science and the choice of using different FPGA or MPSOC boards can have a wildly different effect on the power usage of every implementation shown, with every implemented NN having a completely different objective in mind for object detection with completely different types of input, nor do all off the works selected mention how they got their representative watt usage this makes this comparison somewhat moot. A more true approach

would be to convert the HDL representation into an ASIC and check the power usage of that compared to other works implementing a NN within a ASIC with the same objective of finding the vagus nerve or a similar Bio marker inside of a B-side ultrasound image.

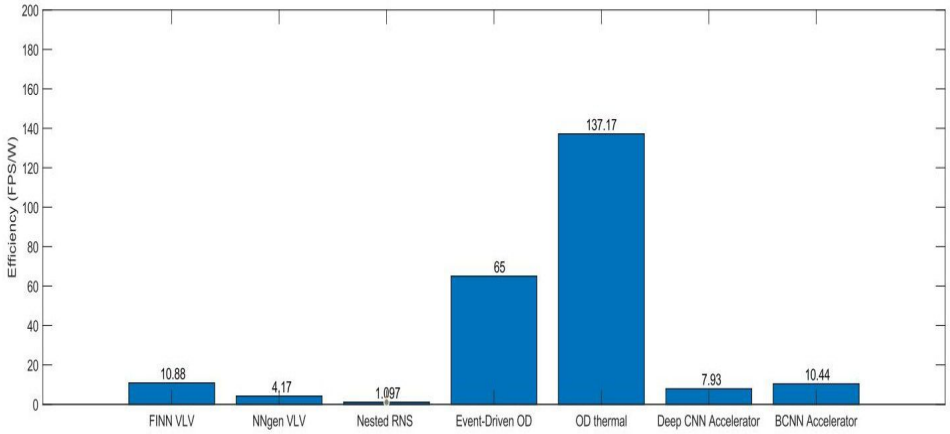


Figure 5.17: A bar chart showing the power usage from real life measurements in watts of the VLV accelerators based on FINN and NNgen within the Ultra96V2.

Despite this assessment certain conclusions can be made by looking at the technical specifications. As can be seen from Figure 5.17 the VLV accelerator made within the FINN framework compares well against all other works except for OD thermal [12] and Event-Driven OD [13], both created by the same team. To explain the difference of efficiency of these outliers a comparison is done to VLV FINN implementation created within this thesis to the most efficient implementation of [12].

Layertype VLV	VlV dimensions	VLV In. F.Size	LayerType [12]	[12] dimension	[12] In. F.Size
Conv. Block	32x3x3x3	224x224	Conv. Layer	64x4x11x11	181×181
Conv. Block	64x32x3x3	112x112	MaxPool	64x64x3x3	43×43
Conv. Block	64x64x3x3	56x56	Conv. Layer	64x64x5x5	22×22
Conv. Block	64x64x3x3	28x28	MaxPool	64x64x3x3	22×22
Conv. Block	64x64x3x3	14x14	Conv. Layer	64x64x3x3	11×11
Conv. Layer	64x64x3x3	7x7	Conv. Layer	64x128x3x3	11×11
Conv. Layer	15x64x3x3	7x7	Conv. Layer	128x128x3x3	11×11
Conn. Layer	1x735	1x735	Conv. Layer	128x128x3x3	11×11
Conn. Layer	7x7x13	7x7x13	Conv. Layer	128x128x1x1	11×11
			Conv. Layer	128x128x1x1	11×11
			Conv. Layer	128x21x1x1	11×11

Table 5.4: Comparison between VLV FINN and [12], note how compared to VLV FINN the linear layers are missing within the work of [12]

Table 5.4 shows the general architectural differences between VLV and OD thermal detector from work [12], do note that a “Conv.Block” consists of a Convolution layer a MaxPool function with a 2x2 kernel, and a ReLu Function while the Conv.Layer just consists of a convolutional layer and a ReLu function. A clear difference between the two implementations is that the OD thermal NN implementation on the FPGA does not consist of a fully connected layer, seen from section 5.3.1 the fully connected layers are the most resource-intensive layer within the VLV FINN implementation as such it can be assumed that they will also be the most energy inefficient and will have to largest inference times. Within the work of [12], these fully connected layers instead run via the user’s laptop making the FPGA implementation resource usage quite lighter and inference speed compared to VLV faster, however, this makes the full setup computer-dependent, making it hard to compare with the VLV FINN implementation in which the full NN is implemented within the FPGA. As such the question arises whether this can be considered a full NN implemented within an FPGA or half of one. A secondary attribute that can be noticed within OD thermal is that in the initial Convolutional and MaxPool layers the input size of the image due to the large kernel sizes has shrunk significantly with the final few layers having a kernel size of 1 making the convolutional process resource light.

Implementation	VLV FINN	[12]
Platform	Ultra96V2	ZCU102
Clock Freq.[GHz]	0.1	0.1
Power[watt]	0.528	1.2
Architecture style	Streaming	Streaming
FPS	5.745	164
Bit width	2-3 bit integer	16-bit floating-point

Table 5.5: Comparison between VLV FINN and [12], note how despite the large increase of FPS over VLV FINN does not result in a linear increase of power requirement.

As can be seen in table 5.5 both the VLV FINN implementation as well the OD thermal use a Streaming Data flow style architecture as defined within section 4.6, while the individual layer architectures are uniquely implemented they do share the same pros and cons compared to a systolic implementation. The OD thermal implementation has a way larger Power usage and a higher FPS compared to the VLV FINN implementation, this thanks to a large amount of parallelism is shown in Table 5.6 resulting in far larger resource usage as shown in table 5.7. Due note that the only way this larger resource usage can be implemented is by using an FPGA with larger resource usage such as the ZCU102, how the power usage within this FPGA is calculated is not defined within the work of [12], which might result in an inaccurate comparison to this work. The bit width within OD thermal has chosen to use PE's using 16 bit floats to increase accuracy, as the VLV FINN implementation uses quantize aware training with automated reduction of bit size while keeping the accuracy to a maximum the VLV implementation can be seen as an objective improvement compared to the implementation by [12].

Layertype VLV	PE VLV FINN	SIMD VLV FINN	LayerType [12]	SIMD [12]
Relu	2	3	Conv. Layer	181
Conv. Block	2	8	MaxPool	x
Conv. Block	2	8	Conv. Layer	22
Conv. Block	2	8	MaxPool	x
Conv. Block	2	8	Conv. Layer	11
Conv. Block	2	8	Conv. Layer	11
Conv. Layer	2	8	Conv. Layer	11
Conv. Layer	3	8	Conv. Layer	11
Conn. Layer	4	15	Conv. Layer	11
Conn. Layer	7	4	Conv. Layer	11
			Conv. Layer	11

Table 5.6: Table comparing the kernel wise parrelisation, versus the pixel wise parrelisation of [12], note that the pixel parrelelism is fully exploited as the parrelism is queal to one axis of the corresponding input shape as detailed in Table 5.4.

Table 5.6 shows a comparison between the FINN VLV implementation and OD thermal implementation in regards to parallelism, it is hard to compare this as there is no industry standard for terminology used for parallelism within NNs, as such what the description of parallelism has been converted to the terminology used within the context of this thesis. The major difference between the VLV FINN implementation and the OD thermal implementation is that VLV FINN uses kernel-based parallelism and OD thermal uses pixel-based parallelism on either the y or x-axis. Using table 5.4 it can be seen that for every layer the maximum amount of pixel parallelism is chosen, as every pixel has its PE for loading the pixel. This massive amount of parallelism is according to the author the major reason why the inference is so fast and the FPS/watts is high.

As can be seen there the OD thermal implementation by table 5.7 uses more resources than the VLV FINN implementation, despite the smaller neural network and the loss of the fully connected layers, due to the high amount of parallelism, this results in higher power usage in watts but also a faster interference. This overall large amount of resources is required raising the question of the overall size of the eventual ASIC and whether it can be implemented within a wearable ultrasound patch as designed within

Implementation	VLV FINN	[12]
LUT	47,298(67.03%)	105,060(38.33%)
LUTRAM	12,588(43.71%)	X
FF	62,774(44.48%)	100,508 (18.34%)
BRAM	50(23,15%)	224 (12.28%)
DSP	90(25%)	390 (15.48%)

Table 5.7: Table showing the resource usage and efficiency in FPS/watt of VLV implemented inside FINN, and [12], as can be seen has a higher resource usage due the large amount of parallelism.

section 4.These variables combined to give a good explanation regarding the extreme efficiency of work [12], especially the high amount of parallelism and the loss of the fully Connected layers gives a good insight into where the next steps can be taken into optimizing the inference as well despite the overall large usage of resources raising the question on the validity of usage within an ASIC as described in this thesis. This work can be used as an example to see how Amdahl's law of increasing parallelism does not only increase interference but also creates a greater power efficiency at a cost of higher resource usage.

Implementation	FINN VLV	[12]	[13]	[14]	[15]	[16]
LUT	47,298 (67.03%)	105,060 (38.33%)	24,672 (9.00%)	427,014 (98%)	N/A	N/A
LUTRAM	12,588 (43.71%)	X	x	0	N/A	N/A
FF	62,774 (44.48%)	100,508 (18.34%)	21,896 (6.30%)	0	N/A	N/A
BRAM	50 (23,15%)	224 (12.28%)	539 (29.5%)	1,235 (42%)	N/A	N/A
DSP	90 (25%)	390 (15.48%)	152 (6.30%)	0	N/A	N/A
Efficiency (FPS/watt)	10.88	137.17	65	1.097	7.93	10.44

Table 5.8: Table showing the resource usage and efficiency in FPS/watt of VLV implemented inside FINN, [12], [13], [14], [15], and [16], as can be seen all the works either have unknown or higher resource usage.

As seen in table 5.8 it is a running theme that all works which out class VLV implemented via FINN on efficiency have resource usages far exceeding this work, the assumption being that these works have implemented a high degree of parallelism, as such due to the aim of creating an autonomous object detector implementation that uses as few resources as possible yet does not require a microcontroller for making calculations, and can be eventually implemented as a ASIC within a patch as described in section 4, FINN VLV does appear as the most optimal choice. Based on these results conclusions can be drawn as described in the next chapter.

6

Conclusion

In this thesis 5 unique works have been created, resulting in to the best of the authors knowledge the first fully automated end to end flow for creating a software NN object detector and converting it towards an HDL representation. This is implemented by using FINN, allowing biomedical engineers without knowledge of digital electronics or Neural networks to simply load in data and run the python files. This resulted in an object detector (Using VLV) within FINN with an accuracy of 87.5 percent on the test set and an efficiency of 10.88 FPS/watt. To create this automated end to end flow, a loss function which keeps the bit width down automatically, working in tandem with the *YOLOLoss* function, to increase the accuracy, was implemented. This resulted in an automated function that creates an implementation of a trained NN with a maximum accuracy and lowest possible bit width combination, trained within one single training session, this is to the authors knowledge the first something like this is created for quantize aware training. To further optimize the object detection process a special NN was created specifically for finding the vagus nerve inside of an ultrasound B-mode image, these factors combined show that NN object detector accelerators implemented within an FPGA do not require a large number of resources, even for streaming dataflow architectures.
Another object detector accelerator implemented within an FPGA using VLV was created within NNgen resulting in a systolic array accelerator with an accuracy of 57.6 percent on the test set with the 4.17 FPS/watt, within the NNgen framework an algorithm was created for accurate quantisation from floating-point representation towards integer representation.
Finally, within the FPGA a cross-correlation template matching implementation was created from scratch using Verilog, this ran with an inference of 0.776 seconds and an accuracy of 27.32 with an estimated power usage of 0.448 watts.
Template matching and VLV have also been represented within an android phone connected via Bluetooth with a Tiny Pico, proving that Bluetooth can be used to send medical data lossless from one device to the other, however, it also has shown to be a very slow medium of transporting information uncompressed, resulting in an overall throughput of 7.06 seconds and an accuracy on the test set of 99.21 percent, finally, another an-

droid phone implementation was created this time focussing on template matching connected via Bluetooth with a Tiny Pico resulting in an overall throughput of 7.5 seconds and an accuracy on the test set of 27.39 percent

Within this thesis, multiple approaches were shown, for the machine learning implementations a very limited data set had to be used. For future work, it would be important to go beyond a proof-of-concept and to create a data set with which a NN can be trained more reliably. To create an as accurate dataset as possible, the B-mode ultrasound images used for training and test data sets should be created via the ultrasound transducer array used within the final designs described within section 4. The created image set should be a variable as possible, with the test subjects consisting of varied ages, races, and gender, the position of the annotations of the vagus nerve also has to be as varied as possible, resulting in a more realistic implementation and accuracy estimation of the implementations. The NN that was created within this thesis, VLV, could be more optimised as the final two fully connected layers are assigned for three classes expecting two objects within a single grid cell, as is however the case within this thesis the only objective is to find the vagus nerve which only appears singularly within an ultrasound B-mode image ergo it also only appears once within a grid cell, as such the last two fully connected layers can be downscaled to expect one class with one object within a gridcell. This should lead to a large reduction in resource usage as can be seen in section 5.3.1, the fully connected layers were seen as the main culprits in resource usage. Finally, an implementation of an active contour model on a cut-out of the found bounding box might increase the overall accuracy of the postprocessing.

The implementation of VLV within NNgen has a very low accuracy through the quantization process compared to the floating-point representation original and the quantize aware model trained in Brevitas. To get a higher accuracy two options can be implemented: Automation of the process described within section 4.5.2 by averaging the error over the entire dataset and implementing it within Pytorch, and implementing an NNGEN representation of VLV and loading it with the parameters taken from a trained Brevitas model. Within the NNgen small research should also be done to see the connection between channel and pixel parallelism and the increase of power/resource usage versus a better FPS.

To get a better comparison of accuracy of VLV versus other object detection models, once a better dataset is created a comparison should be made using the accuracy assessment algorithm detailed in section 3.3.2.

The implementation of VLV within FINN has multiple small issues which decrease its overall performance, as can be seen from the power sweep done in section 5.3.1 of the loss function the average bit width can be set to 2 bit while right now the overall bit width is 2.24 this should result in better performance. Another increase in performance can be by increasing the parallelisation constants of the first two layers of the VLV implementation resulting in faster throughput with no effect on the amount BRAM used as the BRAM usage efficiency lies at or below 50 percent. Finally, the Brevitas implementation can be trained to higher accuracy, right now it is trained at an accuracy of 95.6 percent on the training set but it should be able to reach 99.4 percent of the original Pytorch implementation as shown in section 5.1.

The main problem with the implementation of template matching inside of an FPGA is

the reduced performance as well as the extra resource usage from the LUT block containing a cut-out of the target image and the template. A new design is proposed which will replace the 2 900 long 8 bit LUT blocks with a single 2 8 bit LUT block. Instead of writing to these memory blocks first using the *writesignal* input and then following the calculation function activated via the *go* input a single input is created, this input still expects a bit by bit streaming input but the moment that it fills both of the 8 bit LUT blocks it will automatically calculate the initial denominator and nominator values, after which the LUT blocks will be emptied allowing the cycle to repeat itself. This should reduce a large amount of the LUT usage resulting in a lower resource usage overall and decrease the time taken to calculate the cross-correlation value of a single pixel by one-third as seen in section 5.3.2. Finally, an implementation for a data streaming interface should be made between the Template matching computation unit and the ZYNQ Ultrascale+ MPSoC for easy prototyping.
The major problem of both the template matching and the VLV implementations within an Android phone is to transfer the information from the Tiny Pico microcontroller to the Android phone as can be seen in section 5.2.2, to increase the speed of the throughput two different approaches can be taken: Realise the transfer of the information from the Tiny Pico to the android within a different medium then Bluetooth, such as Direct Wi-Fi, or decrease the size of the transferred information by using compression algorithms such as run-length encoding, area image compression, and Huffman coding. The only real requirement set to this method is that the compression methods are lossless.
To complete the conversion from a research to a product, a conversion from an FPGA implementation to an ASIC should be made, as referred to as the 'imagine algorithm chip' within section 4. The major difference between that proposed implementation and the implementation suggested now is that instead of the output of the transceiver chip going through the microcontroller for pre-processing and then through the ASIC, the pre-processing will be done via a separate chip directly in contact with the ASIC and only the output of the ASIC will be sent to the microcontroller for post-processing. Once an ASIC is created a comparison should be made between the existing ASIC's created by Google specifically for accelerating neural networks.

Finally once the ASIC has been chosen and created the designs described in 4 should be created.

Appendices

A

ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANNs) are statistical models that are partially inspired by and modelled after biological neural networks(NNs). Nonlinear correlation between distinct inputs and outputs can be computed using ANNs. They are powerful instruments that can assist in the solution of problems that cannot be handled using an algorithmic approach. Neural networks can be used to handle problems like image categorization and natural language recognition.

A.1. NEURONS

An ANN will have nodes that are connected to several other nodes, just like a biological neural network. There are many inputs to a neuron, but only one output. The sum of all weighted inputs plus a starting constant is taken into account by a neuron. The output is then passed through a neuron's activation function. A further explanation of an activation function will be given in paragraph A.5. The neuron's output is subsequently passed on to the ANN's next neuron.

A.2. WEIGHTS

A synapse is a link between two nodes. Each synapse has a certain amount of weight. The weight indicates how much the previous node's input should be taken into account. Lower weights imply that the input is less important for the output, whereas higher weights indicate that the input is more important.

A.3. OUTPUT NODES

An ANN's output nodes represent the output node's expected data. The output of an ANN are also called dependant variables. Depending on the data, an ANN can have one or multiple output nodes.

A

A.4. HIDDEN LAYER

An Artificial Neural Network's hidden layer is made up of a network of neurons that connects the input and output layers. Each node in the preceding layer is connected to each neuron in the hidden layer. A single neuron or a group of neurons can make up a hidden layer. There is no hard and fast rule for the number of neurons that make up a hidden layer. In addition, ANNs can have one or more hidden layers. Figure A.1 depicts an example of a rudimentary artificial neural network.

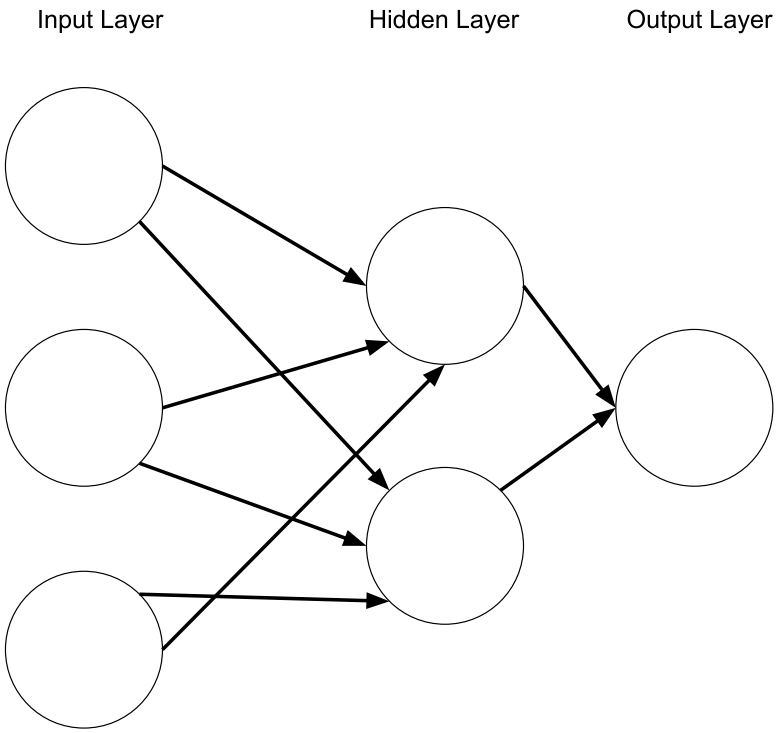


Figure A.1: Figure depicting a simple neural network

A.5. ACTIVATION FUNCTION

A neuron in an ANN does not just send the weighted inputs to the next node. Instead, a function termed an activation function is applied to a neuron's input. The activation function takes a neuron's weighted inputs and transforms them again before outputting them, with the ANN gaining nonlinearity as a result of this change. There are several activation functions available, with LeakyReLu and ReLu being two of the most popular activation functions.

A.5.1. RELU

The ReLu is the most used activation function right now.The ReLU activation function is defined in formula A.1.

$$f(x) = \begin{matrix} x\ if\ x > 0 \\ 0\ otherwise \end{matrix} \tag{A.1}$$

the function sets any negative value to zero, while letting all positives value through unchanged. The ReLu function has a range from 0 to ∞ as seen in the graph Figure A.2.

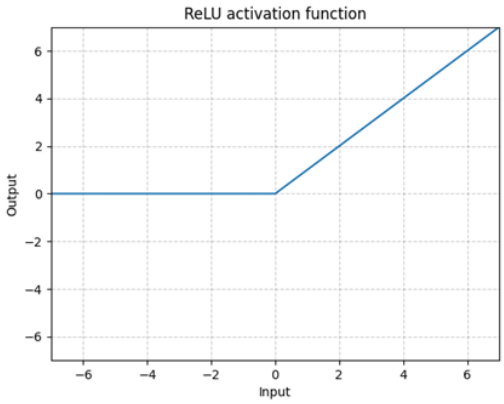


Figure A.2: Graph depicting the ReLu activation function

The main issue is that all the negative values become zero, which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

A.5.2. LEAKY RELU

Leaky ReLu is an attempt of solving the ReLu the negative value problem inherent in the ReLu activation. The function differentiates compared to the regular ReLu that it incorporates a negative element in order to increase the range of the function resulting in formula A.2.

$$f(x) = \begin{matrix} x\ if\ x > 0 \\ bx\ otherwise \end{matrix} \tag{A.2}$$

The variable b usually is 0.01 resulting in the range of -∞ to ∞ represented in the graph of Figure A.3.

A.6. TRAINING AN ANN

An Artificial Neural Network (ANN) must first be trained before it can be utilized to generate predictions. Training refers to supervised learning in the context of this paragraph.

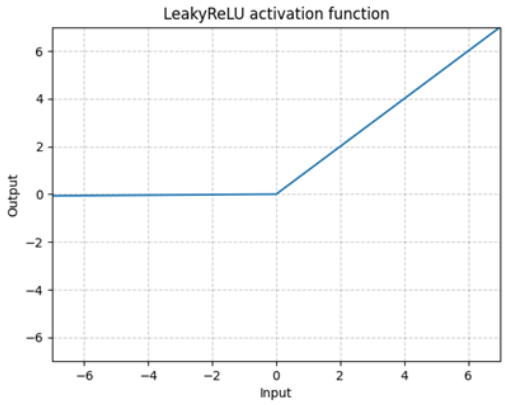


Figure A.3: Graph depicting the LeakyReLu activation function

A dataset that is reflective of the problem to be solved is required to train an ANN. The steps for training an ANN are as follows.

1. Initialization of the weights and the constants of the ANN
2. Forward propagation
3. Evaluation of the loss function
4. Backwards propagation

The ANN must be trained before it can be used. The ANN's weights and constants must be established first. This can be accomplished in a variety of ways. The initial constants and weights can all be set to zero or another uniform number. However, the most frequent technique to establish the weights is to do it at random. The dataset is fed through the ANN once the weights have been established.

A.6.1. FORWARD PROPAGATION

The training of the ANN can commence when the weights of the ANN have been initialized. First, enter the dataset's independent variables and examine the ANN's output. As an example, consider the following dataset.

$$x = [1.2, 0.0, 2.4] \, y = 0.33 \tag{A.3}$$

A single point where x represents a set of independent variables from a dataset and y represents the dependent variable that the ANN is attempting to predict. The independent variables will be entered into the ANN for the time being to see what it predicts in its current condition. The first neuron's output is determined by adding the weighted inputs together and then using the sigmoid activation function to calculate the outcome. This is done as followed:

$$z = (0.7 * 1.2) + (0.3 * 0.0) + (-0.4 * 2.4) + (-0.8) \Rightarrow -0.9 \tag{A.4}$$

$$\sigma(-0.9) = 0.28 \tag{A.5}$$

The calculation of the second neuron is the same as the first neuron which is as followed:

$$z = (0.2 \cdot 1.2) + (-0.2 \cdot 0.0) + (0.3 \cdot 2.4) \Rightarrow 0.5 \tag{A.6}$$

$$\sigma(0.5) = 0.61 \tag{A.7}$$

And finally the output node is calculated as followed:

$$z = (0.5 \cdot 0.28) + (0.4 \cdot 0.61) + 0.9 \Rightarrow 1.3 \tag{A.8}$$

$$\sigma(1.3) = 0.78 \tag{A.9}$$

Unfortunately the calculated output differs from the real output of the dataset. To correct this, the performance of the ANN must be evaluated. This is done with a loss function

A.6.2. LOSS FUNCTION

A loss function, also known as a cost function or error function, is a function that calculates the difference between expected and projected outputs. The loss function is used to compare the output of an ANN to the real value. The performance of the ANN may be evaluated using a variety of loss functions. The following is an example of a loss function:

$$E(y, \hat{y}) = \int_{i=1}^{n} (\frac{1}{2})(\hat{y} - y)^2 \tag{A.10}$$

The y in formula A.10 represents the dataset's anticipated output value, whereas the ŷ is the ANN's predicted variable. The Σ in the formula denotes the total of all the values. The total of all outcomes and projected values in the dataset is used in this situation. In this formula, the square serves two functions. The first is to get absolute values, as just the disparities between the two numbers are required. The second goal of squaring the difference is to punish large discrepancies in the two variables while rewarding tiny differences. The ANN will be trained using the loss function. An ANN that can reliably predict the output based on the dataset is trained by minimizing the output of the loss function. By applying the loss function to the predicted and expected output the following error is calculated:

$$(\frac{1}{2})(0.33 - 0.78)^2 = 0.101 \tag{A.11}$$

The next step is to move backwards and adjust the weights to minimize the loss.

A.6.3. BACKWARDS PROPAGATION

Moving backwards through the ANN and modifying each weight is known as backwards propagation. After tweaking the weights, forward propagation is used once more to see the ANN's output. The loss function is then applied once more to check how the ANN performed. This procedure can be repeated indefinitely. When this procedure is complete, the weights that result in the smallest loss are chosen. The term for this way of training an ANN is brute forcing. It is not just possible that this process will take a lengthy

time to finish. When dealing with increasingly intricate ANNs, however, the time it takes to compute the best set of weights skyrockets. A different method must be employed to fast approach the point of minimal error in order to speed up the process of training an ANN. This can be done with an optimizing strategy which minimizes the loss function. In Figure A.4 a scatterplot of what this brute force method can be seen.

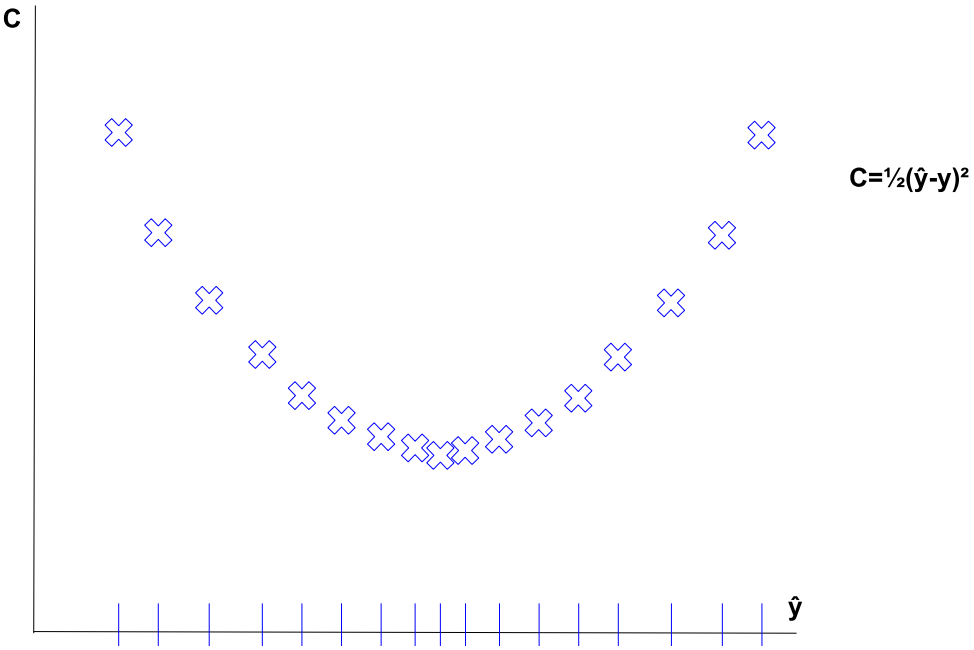


Figure A.4: Graph depicting brute force method of finding minimum error within a function

After many iterations of forward and backwards propagation, the ideal weights for the dataset have been found. In Figure A.5 a visualization of the correct weights and a calculation of the output can be seen.

$$Neuron\,1 = \sigma((-0.1 \cdot 1.2) + (0.1 \cdot 0.0) + (-0.9 \cdot 2.4) \pm 0.5) = 0.06 \tag{A.12}$$

$$Neuron\,2 = ((0.0 \cdot 1.2) + (0.9 \cdot 0.0) + (-0.5 \cdot 2.4) \pm 0.8) = 0.12 \tag{A.13}$$

$$Output = ((-0.6 \cdot 0.06) + (0.20 \cdot 0.12) \pm 0.7) = 0.33 \tag{A.14}$$

A.7. OPTIMIZER

Optimizers, also known as optimizing algorithms, aid in the reduction of the loss function. The optimizer achieves the minimal point of a loss function faster than brute forc-

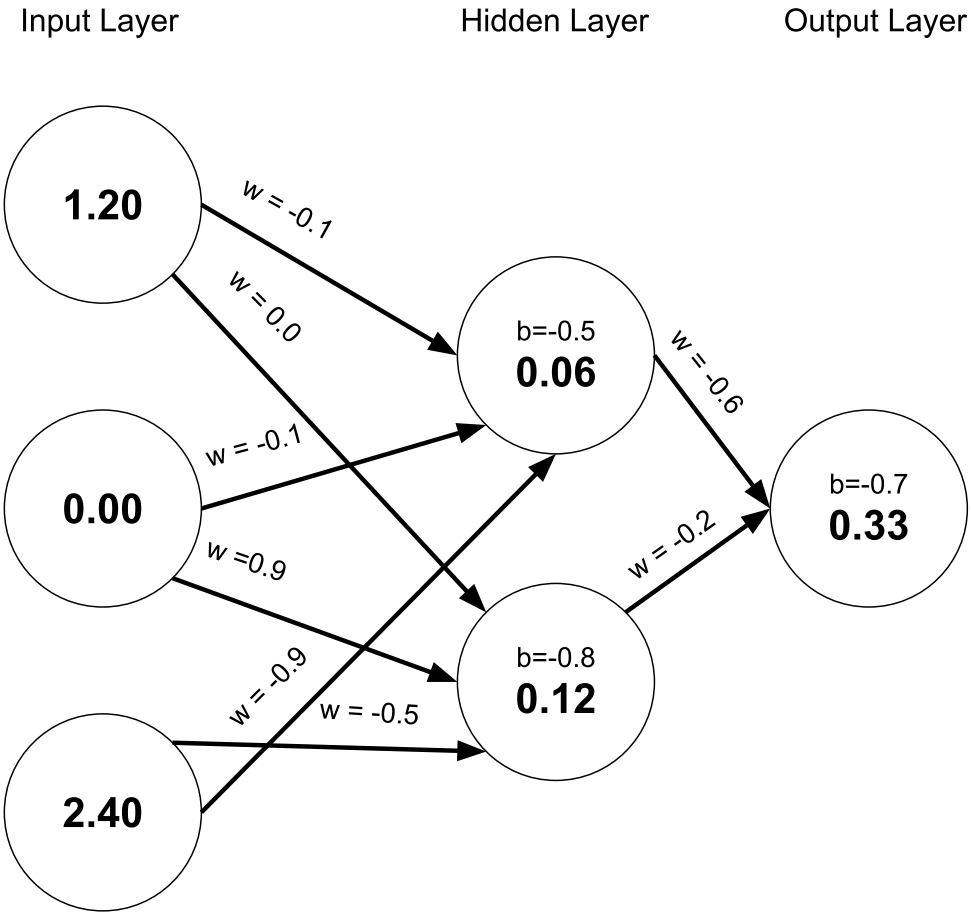


Figure A.5: Figure depicting the weights and calculations of a simple neural network

ing a solution for the ANN. There are a plethora of optimization techniques available, one of which is the gradient descent method.

A.7.1. GRADIENT DESCENT

One of the most significant optimization strategies for training and optimizing an ANN is gradient descent. A multidimensional derivative (dy/dx) of a function is called a function's gradient. A gradient is a multidimensional slope that is calculated instead of a single value that represents the slope of a tangent line. The goal of gradient descent is to use the gradient to approach the minimum of the loss function described in paragraph A.6.2. The first step is to compute the derivative of the first point. Second, the partial derivative is used to compute the contribution of each weight to the loss function. Depending on the slope of the derivative, the weight is modified correspondingly. This approach is continued until the loss function's minimum is found. Figure A.6 shows

a visual representation of the Gradient Descent.

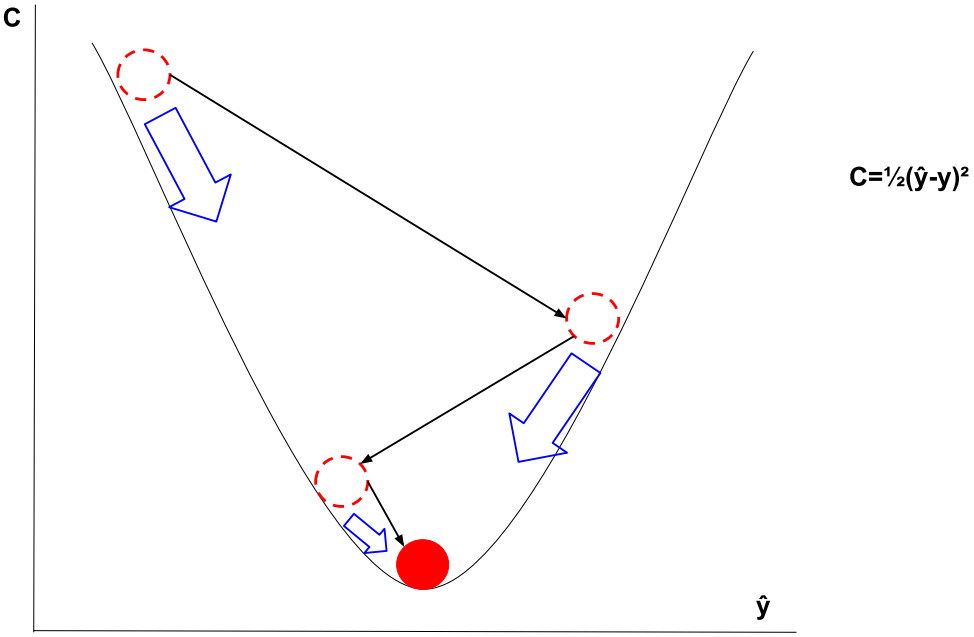


Figure A.6: Figure depicting gradient Descent within a 2d graph

A.8. STOCHASTIC GRADIENT DESCENT

Gradient Descent is a good way to reach a minima of the loss function the problem arises when the loss function has more than one minima. With Gradient Descent the chance of getting stuck in a local minima is big when the loss function is not parabolic. In Figure A.7 the effect of using gradient descent on a function with more local minima's can be seen.

To avoid this, the weights might be modified after a single sample rather than after passing through the entire dataset. This reduces the likelihood of the optimizer becoming trapped in a local minimum. Adjusting the weights after each point in the dataset might take a long time, depending on the size of the dataset. As a result, rather of executing this for each every point in the dataset, it is suggested that you do it in batches. This keeps the optimizer from becoming trapped in a local minimum while reducing the amount of time it takes to modify the weights.

A.9. IMPORTANT LAYERS

To find the Vagus nerve within a ultrasound b-mode image the following specific layers are used.

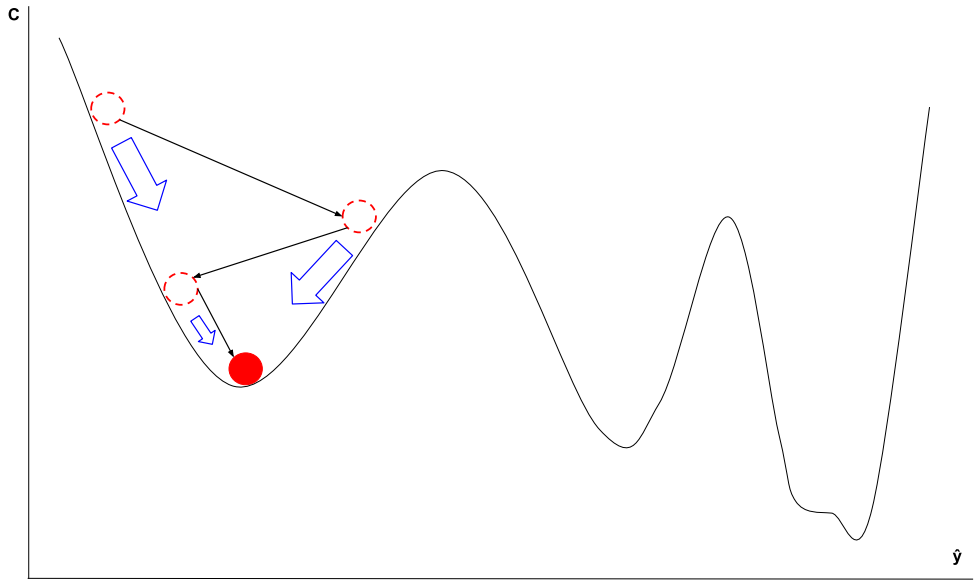


Figure A.7: Figure depicting Stochastic gradient Descent within a 2d graph

A.9.1. CONVOLUTION

Convolution, also known as feature mapping is when a kernel is used to detect how distinctly a certain feature are present in an image. This is done by laying the kernel at the start of a matrix of the image and then multiplying the matrices with each other. After the multiplication, each number of the resulting matrix is summed up. A high number means that the feature is strongly present in the current spot of the image, while a low number means that the feature was barely present in the image. This number is placed in a new matrix. The next step is to move the kernel one stride to left and repeat the process until the kernel has passed the whole matrix. The result of this process is a new matrix called a feature map. The feature map shows where a feature is present in a picture. It also shows how distinctly the feature is present. In Figure D.2 a simplified version of this convolutional process is shown.
The convolutional layer consists of parameters discussed in the following paragraphs

KERNEL

A Kernel which is also known as a filter, feature or a feature detector. As the word “feature detector” already implies a kernel is used to detect features of picture. A kernel is a matrix where the depth is the same as the depth of the input picture and where the height and width can be any size as long as it is smaller than the input picture. The kernels are the weights of the CNN, this means that while training a CNN, the kernels will be adjusted to make the CNN more accurate.

PADDING

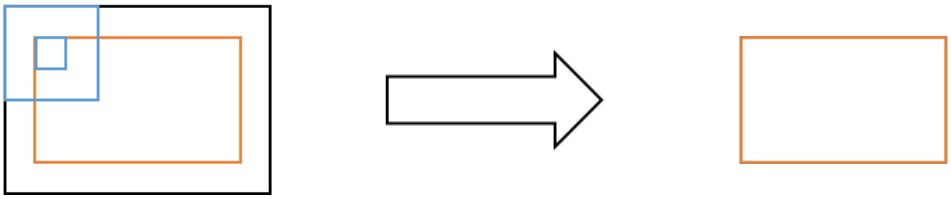


Figure A.8: Figure depicting the "Border effects problem

Depending on the size of the kernel not every pixel will be cantered causing a loss of information at the edge of an image, this is referred to as the “Border effects problem”, within deep neural network with a lot of layers this will result in simply running out of data. In Figure A.8 the Border effects problem is illustrated, as the 3 by 3 kernel cannot be centred on the outer edge the outer layer gets lost. In Figure A.9 it is illustrated how to counteract this problem. Pixel values can be added to the edge of the input data, these pixels are usually valued at 0 in order to not affect the output of the Kernel

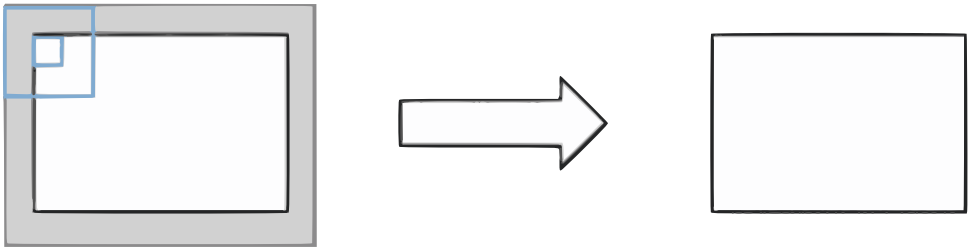


Figure A.9: Figure depicting the negation of the "Border effects problem by adding padding to the model

STRIDE

Stride is the amount of steps taken by the kernel by the kernel for cross correlation.

AMOUNT OF KERNELS

In an effort to capture multiple patterns within an image multiple kernels can be used, a general rule of thumb dictates that the complex the image the more Kernels should be used.

A.9.2. MAX POOLING

Pooling also referred to as down sampling, is the act of reducing the input matrix resulting in an abstract representation of the original matrix. This is done for two reasons. The first being to prevent overfitting and the second is to reduce the amount of input which in turn also reduces the computational cost of the CNN by reducing the amount of parameters. When applying max pooling to a matrix, a smaller two dimensional matrix also called a filter will traverse the input matrix. The filter will usually move by the same

amount as its width. At each stride the maximum number within the filter is included in a new representation map. For a visual representation of max pooling, as seen in Figure A.10. The idea behind max pooling is that when a feature is strongly present in an area of picture the surrounding area is less important and can be ignored.

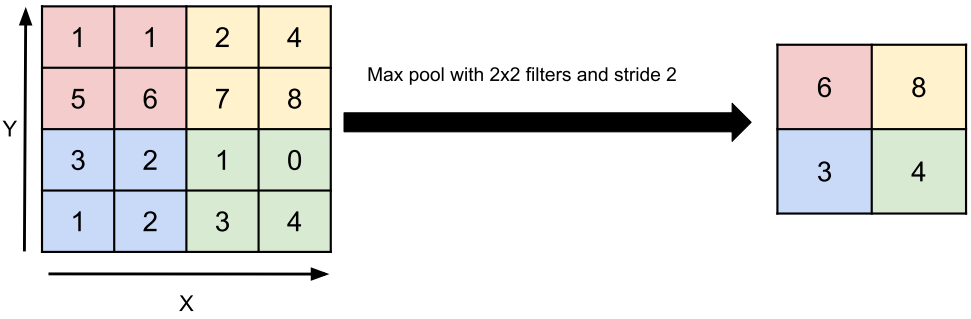


Figure A.10: Figure depicting a visual representation of max pooling

A.9.3. FLATTENING

The result of applying one or more convolutional and pooling operation are multiple smaller matrices. These matrices are then flattened and then used as the input parameters for an ANN. The ANN uses these inputs to classify image.

A.9.4. LINEAR

A linear layer applies transform which can be described as:

$$Y = XA^t + B \tag{A.15}$$

A Feed-forward layer is a combination of a linear layer and a bias. It is capable of learning an offset and a rate of correlation. Mathematically speaking, it represents an equation of a line. Within a neural network linear layers are used in the first layers to scale the input and in the last layers to scale the output. As can be seen in Figure A.11

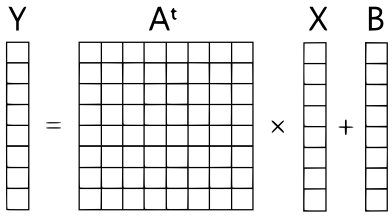


Figure A.11: Figure depicting a visual representation of a linear layer

B

PARALLELISM NEURAL NETWORKS FPGA ACCELERATORS

Parallelism of Neural networks FPGA accelerators can be roughly divided under the following types parallelism:

1. Loop parallelism
2. Data parallelism
3. Task parallelism

How these are implemented is discussed within the following paragraphs.
To show how parallelism is implemented a simple streaming dataflow architecture is created, as shown in Figure B.1, do note that Data parallelism and Loop parallelism can also be implemented within a systolic array implementation.

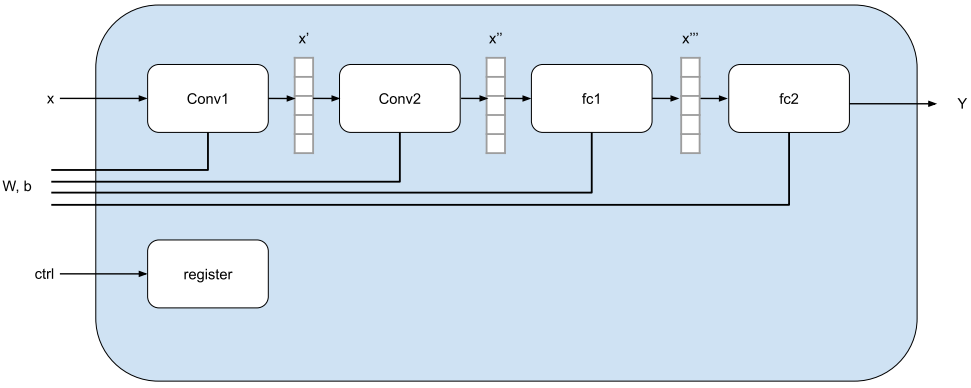


Figure B.1: depiction of a standard streaming dataflow architecture neural network implementation

In this architecture , conv1, conv2, fc1, fc2 are all implemented as different circuits. An SRAM buffer (x', x", x"') inside the FPGA is inserted between each layer, and this buffer serves as input and output for each layer. From this point onwards each layer will be seen as a task within the implementation.

B.0.1. BASELINE SEQUENTIAL PROCESSING

Figure B.2 shows the visualization of the execution time when inference processing is performed 3 times within this architectures.

The execution time for each task is based on the actual operation waveform of the individual task resulting throughput times in which from order of longest to shortest can be seen as conv2 > conv1 > fc1 > fc2. Although conv1, conv2, fc1, fc2 are implemented as separate tasks in this module, these tasks can only run one at a time. Therefore, if the execution time of conv1= t0, conv2= t1, fc1= t2, fc2= t3, the overall execution time over all the evaluations can be seen as 3 * (t0 + t1 + t2 + t3)

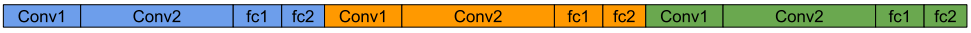


Figure B.2: Fully linear execution of neural network layer tasks

B.0.2. PARALLEL PROCESSING OF TASKS

In this hypothesis the tasks are modified so that they could run at the same time, resulting in multiple tasks processing different interferences at the same time. The execution of this hypothetical is visualised in Figure B.3.

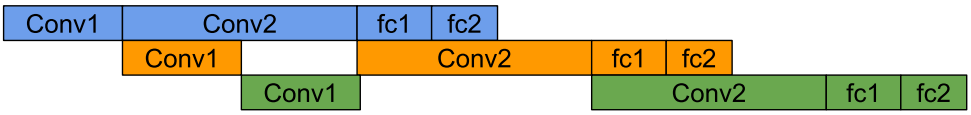


Figure B.3: Parallelized execution of neural network layer tasks

Extracting parallelism so that multiple tasks move at the same time in this way is called task parallelization . Since the execution time of conv2 is dominant in this processing, the processing time for 3 frames is t0 + 3 * t1 + t2 + t3.

B.0.3. IDEAL TASK PARALLELISM

Finally, an ideal version of task parallelism is hypothesized. As mentioned previously, simply extracting task parallelism will make the slowest task the bottleneck, and the overall processing speed will be dictated by the performance of that task. Therefore, the most efficient task parallelism is when all tasks have the same execution time.

In this case, the processing time t0 + 3 * t1 + t2 + t3 remains the same, but t0 = t1 = t2 = t3 the execution time of each task is adjusted so that it becomes , so the performance is improved.

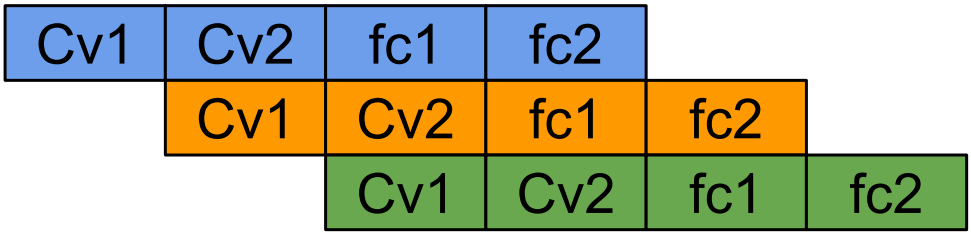


Figure B.4: Ideal parallelized execution of neural network layer tasks

B.0.4. TASK PARALLELISM

The main reason the architecture shown in Figure B.1 cannot simply implement task parallelism is that the buffers cannot read and write at the same time in between tasks, for task parallelism to be achieved this is a necessity.
In this architecture, task-level parallelization is achieved by using the ping-pong buffer x' as the buffer between tasks. The ping-pong buffer is a buffer that prepares two buffers, one for writing and one for reading. The block diagram with the ping-pong is shown Figure B.5.

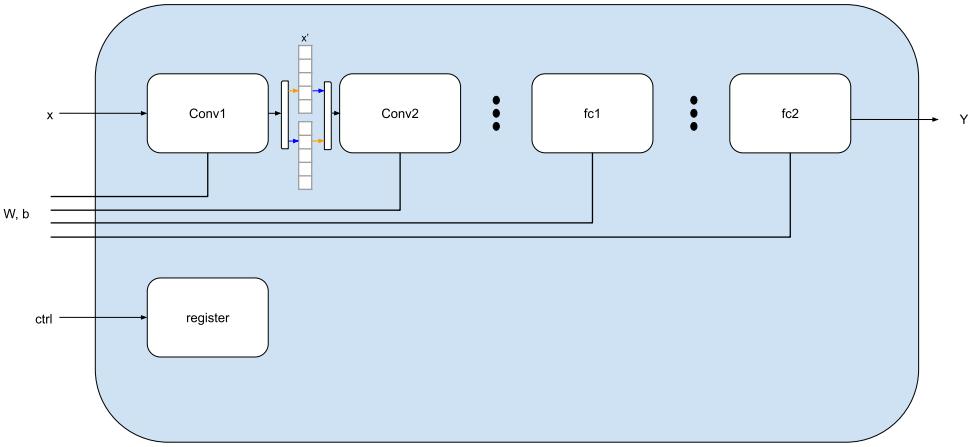


Figure B.5: Depiction of streaming dataflow architecture with implementation of ping pong buffer between layers for task parallelism

When the circuit is configured in this way, the buffer that stores the output from conv1 and the buffer that conv2 reads the input are separate, so conv1 and conv2 can operate at the same time. Although omitted in the figure, all layers can be moved at the same time by performing double buffering for conv2 <-> fc1, fc1 <-> fc2 as well.
Do note that for optimal performance input data as well as weights and biases should be loaded via BRAM then for example externally. BRAM inside FPGAs can operate much faster than DRAM, and data can be read and written stably every cycle. Therefore a more realistic block diagram of a circuit is shown in Figure B.6, within implementation, all the

data such as images and weight sizes are copied in advance inside the FPGA, and each layer is modified to read the data from the memory inside the FPGA.

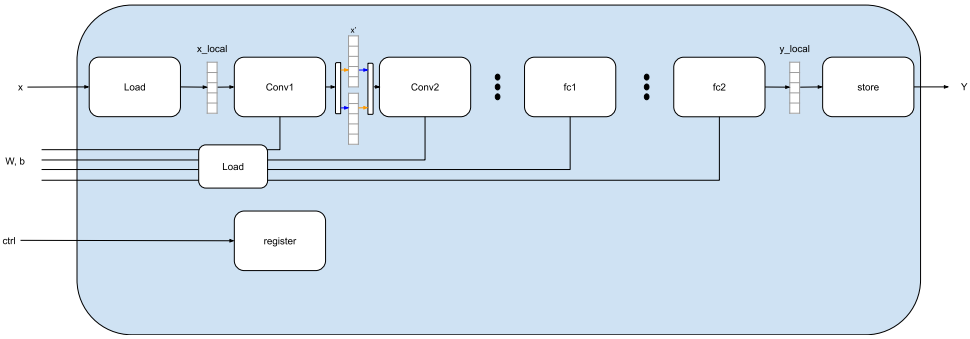


Figure B.6: Depiction of streaming dataflow architecture neural network implementation with implementation BRAM buffers for high throughput loading of inputs and weights.

X_local has the role of temporarily storing the input data from the DRAM from the Load task. The same applies for the weights and biases, for the output layer fc2 the data is buffered in the local memory y_local and later stored in the store layer.
In this figure, the local buffer is a single buffer for simplicity. This buffer should be a ping-pong buffer for task parallelism as described previously.

SPEEDING UP THE CONVOLUTION LAYER BY LOOP PARALLELIZATION

The innermost loop of the convolution function is roughly divided into the following three processes.

1. Pixel, weight load
2. Pixel, weight multiplication
3. Add the multiplication result to the sum register

Figure B.7 below shows a rough visualisation of the above kernel processing flow.

Figure B.7: process flow of standard kernel processing flow of convolution layer

Here, the Figure is based on the assumption that the load processing takes 1 cycle, the fmul processing takes 3 cycles, and the fadd processing takes 4 cycles. With 'i' representing the number of iterations.
Just like in task parallelism we can imagine a hypothetical situation in which the kernel acts almost completely simultaneous and parallel as shown in Figure B.8.

Figure B.8: Ideal parallel implementation of loop parallelism.

Previously, the next iteration started every 8 cycles, but in this hypothetical, the next iteration starts every 1 cycle. Extracting parallelism between different iterations in this way is called loop parallelization . The interval at which iterations can be performed is called Iteration Interval (II), and in this example it is written as II = 1.
In loop parallelism, the way of extracting parallelism is almost the same as in task parallelism. The difference being that task parallelism extracts parallelism between interferences, while loop parallelism extracts parallelism between iterations of processing within each layer. In addition, since the processing of multiple interferences must operate at the same time to extract task parallelism, there are restrictions such as the input data for multiple frames must be expanded to the DRAM on the FPGA in advance. On the other hand, since loop parallelism is completed only within the interference, parallelism can be extracted without any particular restrictions.

Figure B.9: Realistic parallel implementation of loop parallelism due to the fact that the multiplication result to the sum register is dependent on the input of the previous iteration

Performance improvement by duplicating the sum register

As the calculation of every iteration depends on the value within the sum register a more realistic process flow depicted in Figure B.9, as can be immediately concluded gives a huge hit to the performance of the overall design.
A simple solution can be implemented by implementing the amount of sum registers equal to the amount of clock cycles taken, in this case 4 as shown in Figure B.10.

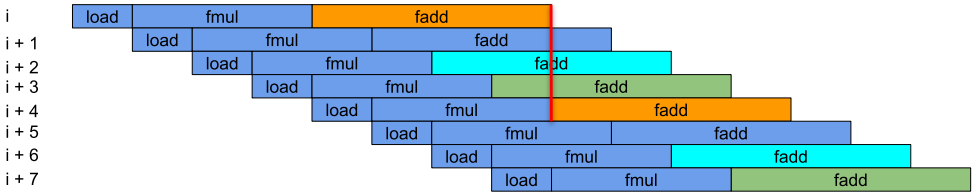


Figure B.10: Ideal parallel implementation of loop parallelism due to multiple sum registers, note that every colour depicts a different sum register.

Within this process flow the separate color of fadd represents a different a different sum

register, as can be seen every 4^{th} iteration of the cycle the same sum register is used again.

B.0.5. DATA PARALLELISM

Data parallelism indicates the parallelism between the data to be processed.
The main difference between loop parallelism and data parallelism is that loop parallelism executes each process in a pipeline as shown in Figure B.11

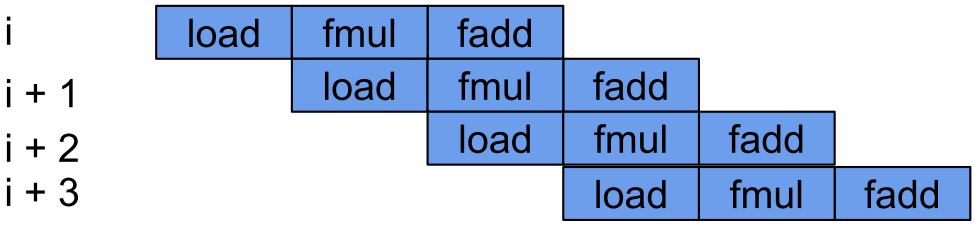


Figure B.11: Parallelized implementation of processing data within a waveform while using purely loop parallelism

Extracting data parallelism, on the other hand, is equivalent to duplicating an processing element (PE). The waveform when two arithmetic units are duplicated is shown in Figure B.12.

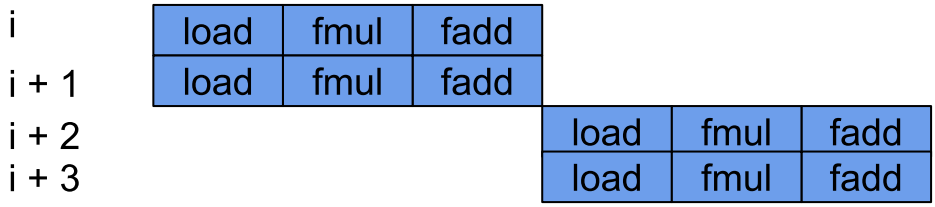


Figure B.12: Parallelised implementation of processing data using two arithmetic units, this is referred to as data parallelism

The difficulty in dealing with data parallelism is that the more data parallelism that is extracted, the more hardware resources it consumes. Of particular concern are memory access resources (load / store). Note that BRAM in the FPGA can usually issue a total of two load / stores per cycle, so if more load/stores per cycle are required, the BRAM will be replicated. While loop and task parallelism is usually implemented within designs as an industry standard due to its low resource usage, data parallelism is something the designer has to choose how to implement whether designing from scratch or using a framework like FINN or NNgen.
Loop parallelism and data parallelism can be extracted at the same time as shown in the Figure B.13.

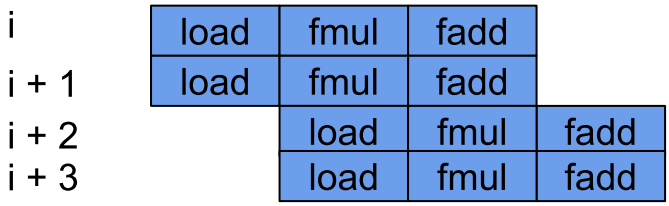


Figure B.13: Parallelized implementation of processing data within a waveform while using both loop parallelism as well as data parallelism

Data parallelism in convolution processing

There are various forms of data processing within the convolution processing loop, within this paragraph two types of data parallelism used within NNgen will be discussed.
The first type of parallelism that can be implemented is pixel parallelism is visualized in Figure B.14.

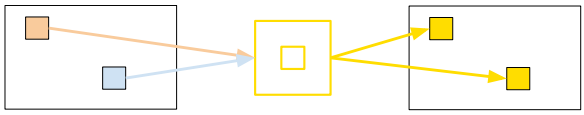


Figure B.14: Convolutional layer pixel parallelism

The orange and blue pixel calculations are independent of each other, so they can be calculated in parallel, with the yellow kernel being used for both calculations. Therefore, the memory access required for one process (two convolutions) is to run the load process twice and the full process once.
The second type parallelism discussed within this thesis is the parallelism between the output channels also known as kernel parallelism shown in Figure B.15.

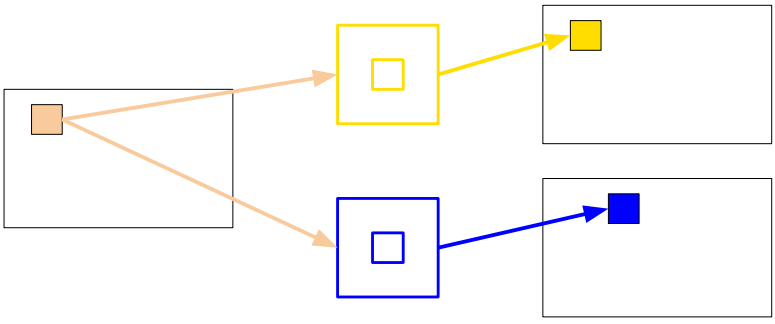


Figure B.15: Convolutional layer Kernel parallelism resulting in data parallelism

Convolution calculations using the yellow and blue kernels do not depend on each other, can be calculated parallel from each other. Which means that in contrast to pixel parallelism, that in kernel parallelism the kernel needs to read twice, but the pixel only needs to be read once. The memory access required for one process (convolution) is to run the load process once and the full process twice.

Figure B.16 shows combined kernel as well as pixel parallelism. Four output values can be calculated with two pixel reads and two kernel reads.

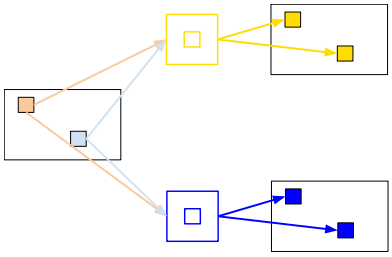


Figure B.16: Both kernel as well as pixel parallelism

The following is a block diagram of the HW to be implemented based on this diagram.

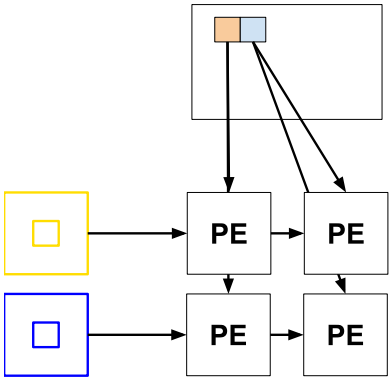


Figure B.17: Pixel parallelism as well as kernel parallelism combined with their used arithmetic units arranged in a grid pattern.

Note that the arithmetic units (PE) can be arranged in a grid pattern.

When data parallelization is performed only for one axis (pixel / output channel), the number of processing elements equal only the amount of parallelism for that axis. As mentioned above, an increase in the degree of parallelism results in an increase in the usage of memory resources such as BRAM. Parallelization on one axis does not use all ports for BRAM/DSPs due to only partially parallelizing the convolutional processing loop.

On the other hand, if the two axes are parallelized as shown Figure B.17, and the degree of parrelelism in both axis are equal the usage of memory access ports should be balanced resulting in optimal resource usage. As such if the designer chooses to increase pixel parallelism it almost always cost minimal resources to also increase kernel parallelism within the same task, vice versa this

C

WORKFLOW NNGEN DETAILED

SETTING UP ENVIRONMENT

The following environment was set up for NNgen usage

1. Ubuntu 1.04.4.2 LTS
2. Python 3.7.7
3. Vivado 2019.1

Within a python virtual environment the following modules have been installed.

1. veriloggen: 1.8.2
2. pyverilog: 1.2.1
3. pillow: 7.1.2
4. onnx: 1.7.0
5. torch: 1.5.0
6. torchvision: 0.6.0
7. jupyter: 1.0.0

When these requirements are met NNgen can be installed using the following commands.
git clone https://github.com/NNgen/nngen.git
cd nngen
git checkout b063ad58052229057864a10b4e80c9d96cf53154
python3 setup.py install
The setup is complete.

CREATE PYTORCH IMPLEMENTATION

Accelerators generated by NNgen do not currently support floating point arithmetic, only integers (or fixed decimals) are available.
Since many FPGAs consume more circuit resources to implement floating-point arithmetic than integer arithmetic, it is preferable to use low-bit width integers from a circuit perspective.
As described within Paragraph 3.5.2 two methods exist for quantisation: Post-Training Quantization) and and Quantization-Aware Training. For this implementation the choice was made to implement Post-Training Quantization, in contrast of Quantization-Aware-training implementation used by Finn.
The neural network described in paragraph 4.2 was implemented within a Pytorch environment detailed within [52].

CONVERT PYTORCH MODEL TO ONNX GRAPH

Pytorch has its own functionality to convert Pytorch models into ONNX models, when converting to ONNX models the nodes within the graphs should be able to be converted to it its NNgen representation. Within git version *b063ad58052229057864a10b4e80c9d96cf53154* the following ONNX nodes are supported.

1. add: Add two tensors
2. sub: Subtraction of two tensors
3. add_n: Add n tensors
4. multiply: multiply two tensors
5. multiply_shared: Multiply two tensors (share multiplier)
6. multiply_add_rshift_clip: After the sum of products (x * y + z) with a tensor of 3, right shift and clip
7. div: division of two tensors
8. neg: sign inversion
9. abs: absolute value
10. equal: Match comparison of two tensors (==)
11. not_equal: Mismatch comparison of two tensors (! =)
12. less: Comparison of two tensors (<)
13. less_equal: Comparison of two tensors (<=)
14. greater: Comparison of two tensors (>)
15. greater_equal: Comparison of two tensors (> =)
16. sign_binary: returns +1 for positive values, -1 for zero or negative values

17. sign_ternary: returns +1 for positive values, -1 for negative values, 0 for zero
18. clip: Clip (saturation) with the maximum and minimum values determined from the bit width
19. where: ternary operator (a? B: c)
20. lshift: left shift
21. rshift: right shift
22. rshift_round: Rounded right shift
23. zeros_imm: returns an immediate value of 0
24. ones_imm: returns an immediate value of 1
25. full_imm: returns any one immediate value
26. zeros_imm_like: Returns zeros_imm with the same shape as an existing tensor, similar to zeros_like
27. ones_imm_like: Returns ones_imm with the same shape as an existing tensor, similar to ones_like
28. full_imm_like: Returns full_imm with the same shape as an existing tensor, similar to full_like
29. relu: ReLU (Rectified Linear Unit)
30. leaky_relu: Leaky ReLU
31. exp: exponential function
32. sigmoid: sigmoid function
33. add: Add two tensors
34. sub: Subtraction of two tensors
35. add_n: Add n tensors
36. multiply: multiply two tensors
37. multiply_shared: Multiply two tensors (share multiplier)
38. multiply_add_rshift_clip: After the sum of products (x * y + z) with a tensor of 3, right shift and clip
39. div: division of two tensors
40. neg: sign inversion
41. abs: absolute value

42. equal: Match comparison of two tensors (==)
43. not_equal: Mismatch comparison of two tensors (! =)
44. less: Comparison of two tensors (<)
45. less_equal: Comparison of two tensors (<=)
46. greater: Comparison of two tensors (>)
47. greater_equal: Comparison of two tensors (> =)
48. sign_binary: returns +1 for positive values, -1 for zero or negative values
49. sign_ternary: returns +1 for positive values, -1 for negative values, 0 for zero
50. clip: Clip (saturation) with the maximum and minimum values determined from the bit width
51. where: ternary operator (a? B: c)
52. lshift: left shift
53. rshift: right shift
54. rshift_round: Rounded right shift
55. zeros_imm: returns an immediate value of 0
56. ones_imm: returns an immediate value of 1
57. full_imm: returns any one immediate value
58. zeros_imm_like: Returns zeros_imm with the same shape as an existing tensor, similar to zeros_like
59. ones_imm_like: Returns ones_imm with the same shape as an existing tensor, similar to ones_like
60. full_imm_like: Returns full_imm with the same shape as an existing tensor, similar to full_like
61. relu: ReLU (Rectified Linear Unit)
62. leaky_relu: Leaky ReLU
63. exp: exponential function
64. sigmoid: sigmoid function
65. reshape: change shape
66. cast: change the data type

67. expand_dims: expand dimensions
68. transpose: transpose
69. conv2d: 2D convolution
70. binary_weight_conv2d: 2D convolution with binarized weights
71. ternary_weight_conv2d: 2D convolution with ternary and t-weights
72. log_weight_conv2d: 2D convolution with log-quantized weights
73. matmul: Matrix product (mainly used for fully connected layers)
74. max_pool: maximum value pooling
75. avg_pool: average pooling
76. max_pool_serial: Lightweight implementation of maximum pooling (available if area size is larger than stride)
77. avg_pool_serial: Lightweight implementation of mean pooling (available if area size is larger than stride)
78. pad: add margins
79. slice_: Cut out a vector from a matrix
80. concat: combine multiple tensors
81. upsampling2d: Upsampling (reverse of pooling)
82. pad: add margins
83. slice_: Cut out a part
84. concat: combine multiple tensors
85. upsampling2d: Upsampling (reverse of pooling)

As ONNX is a open source community project the nodes created through the Pytorch through ONNX change over releases. As such it is recommended to manually select which operator version nodes match the ones supported by NNgen, further details of this are detailed within the ONNX github.

Convert ONNX format model to NNgen format

The first step of converting an ONNX graph into a HLS representation is to convert the ONNX graph into a NNgen graph. This can be done via the *from_onnx* operation. Within this operation the bitwidth for the scaling, the bias, the activation function and the weight, note that the bitwidth for these datatypes across all nodes will be uniform and the ability to select bitwidth for each individual layer is not given. As the resource usage is the main concern within implementing a NN inside of a ultrasound patch the bitwidth for all values is chosen to be the minimal value of 8 bits with an exception of the bias which is given 32 bits space.

QUANTIZE THE NNGEN MODEL

Since NNgen is a compiler that quantizes a trained model, it has a quantisation function for Post-Training Quantization that quantizes the trained model from a floating point implementation towards a integer representation. When quantizing to an integer, it is necessary to appropriately truncate the value by right-shifting so that the calculation result of each layer does not overflow. NNgen's quantizer randomly generates input data and determines the amount of shift based on the expected mean and variance of the input data.

The code used to automatically go through the dataset and find the mean and variance of the input can be found in appendix X.

SET HARDWARE ATTRIBUTES

The hardware parallelism generated by NNgen can be specified in the form of a parameter, separate from the model definition. In this example, the degree of parallelism is specified for each of the conv2d operator (including the matmul operator), pool operator, and element-by-element operation. To keep the amount of resource usage as small as possible there is the bitwidth of the quantisation is kept as low as NNgen allows it, namely 8 bits. To further increase accuracy right-shift operations are inserted to the tail of (almost) each operator. The amount of right-shift also can be assigned via the cshamt_out parameter.

SIMULATING THE NNGEN GRAPH

Before generating the hardware from this computational graph, it's a good to check if the defined computational graph behaves as desired after quantisation.

Computational graphs on NNgen can be run as Python software by passing input data as arguments. Inside, it consists of functions corresponding to each operator that generate the same integer operation results as the hardware. The user can compare this execution result with the calculation result with the original floating point number to check whether it works as expected before hardware conversion.

By passing the input data as an argument, the calculation graph in NNgen format can obtain the same result as the calculation result on the quantisation hardware by software execution. This makes it possible to check how much the behaviour until final recognition changes due to quantisation before hardware conversion.

Within the NNgen software implementation it is possible to manually generate a layer by layer simulation by creating a custom graph in which every layer has its own independent output. if any layers output does not match the original ONNX/Pytorch layer, the amount of right shift should be adjusted within that layer as described in paragraph C. The process to do so is described in Figure C.1

HARDWARE DESCRIPTION GENERATION

To generate a bitfile within an hardware synthesizer an IP needs to be generated. The *to_ipxact* method generates a hardware-structured Verilog HDL description and an IP-XACT format configuration file from the computational graph.

Next to the output of the quantized array a log will be printed out, this log points to all the memory addresses for input, weights and outputs.

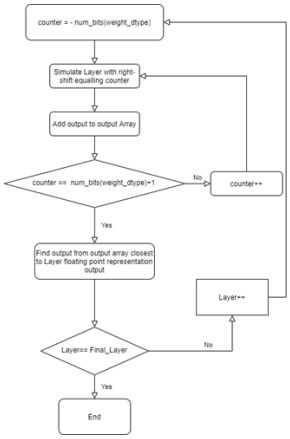


Figure C.1: A flowchart depicting the workflow a user needs to take to create an accurate quantized model compared to the original floating point model

SAVE THE QUANTIZED WEIGHTS

As the HDL generation finishes, the weight parameters of the neural network used on the actual FPGA are converted to the ndarray format of a single numpy array with the *export_ndarray* and saved as a numpy file. In the accelerator generated by NNgen, it is important to predefine the shape of weights used inside of the accelerator. This numpy array will be loaded onto the DRAM of the generated accelerator.

CREATE BITSTREAM

At this point the generated IP can be loaded inside a hardware synthesizer and the constraints can be set for implementation within an FPGA, resulting in a bitstream, however due to the unique property of the Ultra96 being an MPSoC the resulting IP will instead be connected to the Ultra96's ARM processor. To do this first a block diagram has to be created containing the generated NNGen IP. The ARM processor is depicted within the vivado environment as a ZYNQ Ultrascale+ MPSoC IP and should be added together with the NNgen IP.
Using the function *Run Block Automation* a board preset will be set on the processing system, this should add a reset port on the ZYNQ block. NNgen's software repository is provided with a driver for running the NNgen IP connected to the ARM processor using PYNQ, within this driver AXI input/outputs are defined, to sync up this interface the AXI Bus within the PS_PL interface should be defined as:

1. AXI HPM0 FPD
2. AXI HP0 FPD

Any other AXI interface whether it is Master or Slave should be removed from the PS-PL Configuration.
To finalize the block design the following IP need to be added:

1. AXI Interconnect: connects the AXI memory-mapped master ZYNQ IP to the AXI memory mapped slave NNgen IP
2. AXI SmartConnect: connects the AXI memory-mapped master NNgen IP to the AXI memory mapped slave ZYNQ IP
3. Processor System Reset: Gets connected to PL-PS interface's reset port of the ZYNQ IP resulting into an asynchronous reset connected to the NNgen's IP reset port

These connections can be automated via Vivado's *Run Connection Automation*, this should result in the block design shown in Figure C.2.

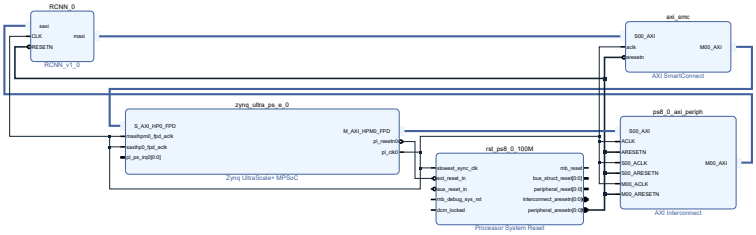


Figure C.2: The created block design after correct implementation of NNgen IP *RCNN_0*

The bitstream at this point can be generated as normal with the *generate Bitstream* function within the Vivado GUI.

RUN THE NNGEN IP WITHIN PYNQ

If the ultra96V2 or any other PYNQ compatible MPSoC device is chosen as the basis to run the generated circuit the following described steps can be taken execute to evaluate the resulting IP.
PYNQ v2.5 is a python package installed in a linux-based OS within the ARM processor and used as an interface with the wider FPGA part. In PYNQ, the memory area shared by the CPU and FPGA circuit is secured by using a mechanism called CMA (Continuous Memory Allocator). Here, this memory area is called the CMA area.
For the created accelerator to work proper the input data, and weight parameters need to be loaded into CMA area in advance via the ARM controller using PYNQ, the addresses in which this data is loaded is described within the log referenced in paragraph C. Once the generated accelerator on the FPGA is executed, input,output, and temporal data is read and written out of the CMA area by the DMA Controller to perform calculations.
Within the pre-built PYNQ Released by Avnet, the maximum size of the CMA area is 128MB.
For larger neural network implemented within NNgen the CMA area might need to be increased. The maximum size of the CMA area can be specified inside the build script of the boot-related files.
A directory is created within the notebooks subdirectory, within this directory the following files are added:

1. *Nngen_ctrl.py* found within the nngen repository
2. Input image
3. Weight file *RCNN.npy* generated in paragraph C
4. The generated *design_1_wrapper.bit* and *design_1.hwh* from paragraph C renamed to the generated IP's name, e.g. *RCCN.bit* and *RCNN.hwh*, respectively

Within this directory a .ipnyb file can be generated, within this file the created accelerator will be ran. The .ipnyb file is found in appendix X with comments annotating the code.

D

WORKFLOW FINN DETAILED

D.0.1. INSTALLING FINN

To use FINN the following were run from a virtual machine

1. Ubuntu 18.04 with bash installed
2. Docker without root
3. A working Vivado 2019.1 or 2020.1 installation
4. A VIVADO_PATH environment variable pointing to the Vivado installation directory (e.g. the directory where settings64.sh is located)

The generated accelerator was run on an Ultra96V2 running PYNQ 2.6.
It is highly advised to run FINN with at least 16 GB of RAM, because synthesizing is a very RAM-intensive procedure. The created environment should at least comprise 150 GB of free storage.

D.0.2. BREVITAS IMPLEMENTATION

The typical start of the conversion process is to create a neural network description in PyTorch and train it with Brevitas. Brevitas is a PyTorch library for quantisation-aware training and allows for exporting models suited for the FINN compiler flow; models are exported in ONNX format with datatype annotations to weights to enable quantizing the weights to datatypes smaller than 8 bit integers.
Brevitas implements a set of building blocks at different levels of abstraction to model a reduced precision hardware data path at training time, additionally, a super-set of quantisation schemes implemented across various frameworks and compilers under a single unified API is supported.
The main objective of the Brevitas PyTorch implementation is to create a quantized representation of the VLV network that will result in an as resource light as possible HDL representation on the FPGA. The resource usage directly correlates to the bit width used

within the individual layers as the target device is relatively small the average bit width over the entire model should be kept as low as possible.
This can be done manually as the bit width in a layer can be manually set via the parameter *weight_bit_width* and *act_bit_width*, note that the default bit width is set to 8 bit. The model can simply be trained in a similar vein as the original PyTorch implementation. To decrease the resource usage the user is expected to manually lower the bit width parameter values for each layer per run while making sure that the overall accuracy of the NN does not decrease. As such if the objective is to decrease resource usage of the VLV implementation with a bit width range of 2 bits to 8 via decrease of bit width of individual layers the theoretical amount of runs needed to find the optimal application would require $6^{(10)} = 60466176$ runs. Where 6 stands for the bit width range and 10 for the number of layers.

As part of this thesis project, a loss function for Brevitas was developed which automatically tries to decrease the overall bit width of the model without losing any form of accuracy, allowing VLV and by extension, any other NN implemented within Brevitas to have a as time-efficient and minimized user input training as possible, while getting a as accurate and resource effective implementation within FINN. Finding a balance between making the model more efficient (i.e. smaller bit width) without sacrificing accuracy is a non-trivial problem. It can be looked at as a multi-objective optimization problem or as a constrained optimization problem.
The various ways of implementing the effects of bit width can be quite varied, the following loss function x has been chosen for its simplicity to implement bit width loss in tandem with the regular VLV loss function.

$$BitLoss = \sum_{e=0}^{n} avgbit_e^c \tag{D.1}$$

With $avgbit$ being defined in formula D.2.

$$avgbit = \frac{(\sum_{e=0}^{n} Bitwidth(e))}{(n)} \tag{D.2}$$

With e being a parameter within VLV and n the total amount of adjustable parameters within VLV.
The variable c should be run through a sweep with each training session starting from scratch to retrieve the optimal size of the accelerator while achieving a precision that is similar to the floating-point representation of VLV. The total loss function of the Brevitas implementation of VLV is summarized in function x

$$\sum_{e=0}^{n} YoloLoss_e + BitLoss_e \tag{D.3}$$

As FINN accepts an ONNX model as input the Brevitas model needs to be converted to the ONNX model. The model is only composed of ONNX standard Nodes, but it also includes additional attributes to the ONNX nodes that can be used to represent low precision data types. To work with the model it is wrapped into *ModelWrapper* provided by FINN.

D.0.3. TIDY-UP AND DEFINE INPUT

The Tidy-up is the first graph transformation. This pre-processing includes input definition. It is common for neural networks to perform some preprocessing on the raw data before it can be used in a machine-learning framework. This may be used to convert 8-bit RGB data into floating-point values between 0 and 1.
These pre-operations in FINN can be baked into an ONNX graph. In some cases, they can be very beneficial for performance because the accelerator can directly consume raw data and not go through CPU preprocessing.
Within the original Brevitas preprocesses VLV input images with *torchvision.transforms.ToTensor()* before training, which converts 8-bit RGB values into floats between 0 and 1 by dividing the input by 255. FINN allows the user to achieve the same effect by exporting a single node ONNX graph to divide by 255. FINN can also set the input tensor to 8 uint so that it knows which level of precision to use.
The Tidy up transformation takes the data types and shapes of the Tensors and extracts them from the model properties. It then sets them in the model's *ValueInfo*. Each graph node has a name attribute that is unique and human-readable.
Constant folding is used to simplify the model. It determines the output of a node that has constant inputs. After determining the output, the result is set as constant-only inputs to the next node. The old node is also removed. Although this transformation changes the structure of the model, it is a transformation that is usually always desired and can be applied to any model.

D.0.4. STREAMING FLOATING

[] The second graph transform is the streamlining transformation. Note that the FINN compiler currently only supports fully quantized NNs. However, in practice, quantized NNs can contain floating-point computations in between quantized layers to improve the accuracy of the model. For example, in the case of the quantized VLV model, batch normalization layers with floating-point parameters for the mean and standard deviation can be found in front of several activation functions as well as channelwise scaling operators after several convolutional layers. These floating-point computations limit the ability to fully leverage the benefits of deploying such a model on FPGA; floating-point computations and floating-point parameters are more expensive in terms of resources and power than their integer counterparts. The streamlining transformation eliminates all floating-point operations from the model [57].
To understand how the floating-point parameters get eliminated, or also referred to as absorbed, the underlying principle of the streamlining transformation must be explained. First, note that the streamlining algorithm only works for uniform quantizers. The algorithm contains three fundamental steps that enable the conversion to a network with only integer operations and parameters:

1. Quantization as successive thresholding
2. Reordering and collapsing linear transformations.
3. Absorbing linear operations into thresholds

QUANTIZATION AS SUCCESSIVE THRESHOLDING

Given a set of threshold values $t = t_0, t_1 \dots t_n$, the successive thresholding function T(x,t), which maps any real number to an integer within the interval [,.n], returns the integer that corresponds to the number of thresholds where x is greater or equal to:

$$T(x,t) = \begin{matrix} 0, & \text{for } x \leq t_0 \\ 1, & \text{for } t_0 < x \leq t_1 \\ \cdots & \cdots \\ n-1, & \text{for } t_{n-2} < x \leq t_{n-1} \\ n, & \text{for } t_{n-1} < x \end{matrix}$$

A uniform quantizer can be expressed by successive thresholding, followed by a linear transform such that Q(x) = a · T (x) + b. For example, the 2-bit uniform HWGQ quantizer could be expressed as HWGQ(x) = 0.538 · T (x, t) $with$ t0 = 0,t1=0.807,t2=1.345. This technique is not economical for activations with more then few bits. The number of thresholds increases exponentially with activation bit width.

D

MOVING AND COLLAPSING LINEAR TRANSFORMATIONS

Any sequence of linear transforms can be rearranged into one linear transformation. First all floating point linear operations can be moved between the quantized matrix operator and activation quantisation. Then, they are all collapsed into one linear transformation. For example The linear transformation ax + b that was used to quantize the activation layer before can be moved beyond the matrix multiplication. Since W · (ax + b) = a · (W x) +W it forms a sequence with the α–scaling, batch normalization, and the linear transformation. This sequence of three linear transforms can then be reduced to one linear transformation.

ABSORBING LINEAR OPERATIONS INTO THRESHOLDS

The final step in the streamlining process is to update the threshold values as ti $\leftarrow \frac{ti-b}{a}$ using the parameters a, b of the linear transformation. Observe that in the inequality t0 < x ≤ t1, x can substituted as ax + b, resulting in the entire formula being rewriten as $\frac{t0 - b}{a} < x \leq \frac{t1 - b}{a}$. This update allows the user to remove floating-point linear transforms completely and feed the result from the quantized matrix operation directly into each subsequent thresholding layer. If the input to the quantized matrix operation is an integer Each threshold can be easily rounded up to get the closest integer, even if the input to the quantized matrix operation is an integer (i.e. the activations of the previous layer were also quantized), without changing the results.

D.0.5. LOWERING CONVOLUTIONS

The third graph transformation is to lower convolutions to matrix multiplications . The operation of a standard convolutional layer can be thought of as a filter sliding over the input data, as explained in Section A.9.1. Within the FINN compiler a convolution operation for each output pixel is obtained by a dot product between a vector of input pixels and a vector of kernel weights. Both depthwise, as well as pointwise convolutions, are supported resulting in different implementations since there are two domains in which lowered convolutions can be executed; software domain (Python, ONNXruntime) and

hardware domain. The lowering in both software level as well as hardware-level is shown in Figure D.1

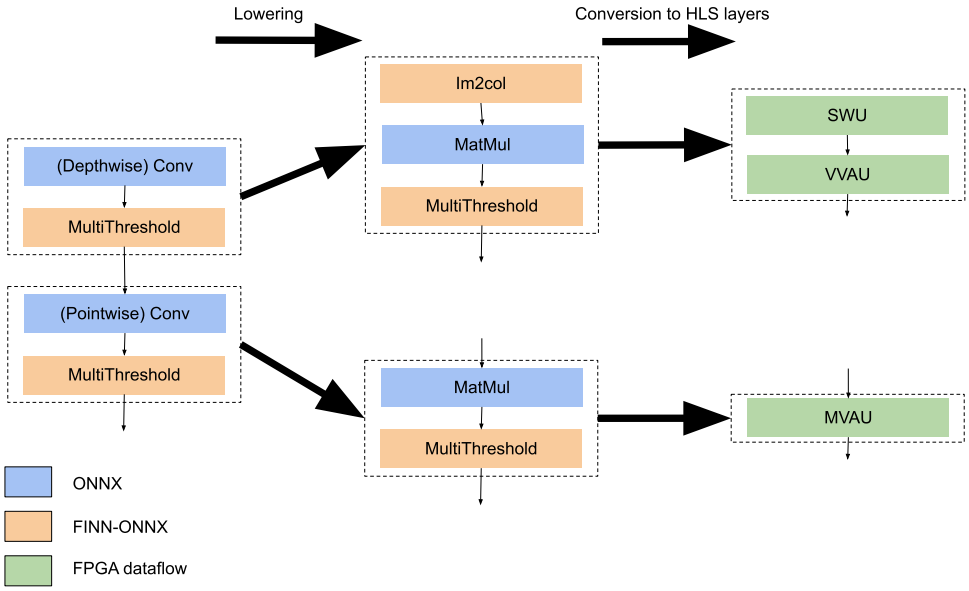


Figure D.1: Schematic of how the ONNX operators are transformed to FPGA dataflow nodes, starting from a regular convolution and applying the lowering transformation and subsequently converting the layers to HLS layers. Blue nodes are standard ONNX operators, orange nodes are FINNONNX operators, and green nodes represent FPGA dataflow nodes.

The first two nodes are used to represent the graph until the layers are converted to HLS layers, as shown in Figure D.1, and can be used used to execute the graph utilizing ONNXruntime. This paragraph will describe depthwise convolution, which is the most common type of convolution layer within VNV. A MatMul operator replaces convolution. The Im2Col operator will be replaced by the SWU one after conversion to HLS layers. Depending on whether a depthwise, regular/pointwise or mixed convolution was applied, the MatMul can be replaced by a Vector Vector Activate Unit (VVAU) or Matrix Vector Activate Unit (MVAU).

In Figure D.2, a functionally reduced pointwise convolutional layer is expressed as matrix multiplication. This example shows the convolution of an input image having three input channels (IFMC), of dimensions [IFMH and IFMW] = 3, 3 and a kernel with dimension [KH or KW] = 2, 2. The input image is not padded and strides and dilation values are assumed to be 1. The SWU organizes the input image into rows, which contain all of the input pixels necessary to calculate a single output. The lower leftmost matrix illustrates this. Each row of this matrix includes, for each output pixel, all input pixels that are within the receptive fields of that particular output pixel. Each vector product is created by rearranging the weights in the correct format.

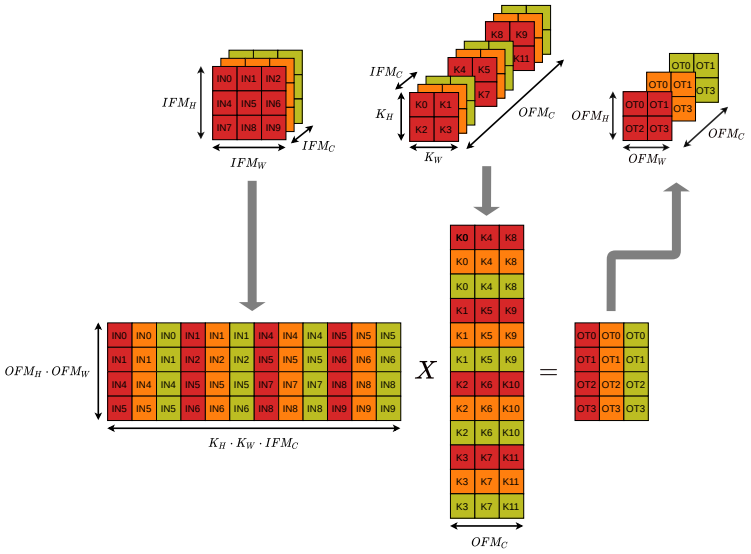


Figure D.2: An example of a convolution lowering method for a regular convolution. Note that the image represents the SWU and matrix multiplication from the perspective of the HLS implementation of these kernels as well as how a lowered convolution is executed in the FINN ONNX domain taken from [7]

Figure D.3 shows a lowered Depthwise layer of convolutional layers. Note that the Depthwise convolution requires significantly fewer parameters and multiply-and-additions than the regular convolution.

D.0.6. CONVERTING TO HLS LAYERS

Before the layers are folded, it is necessary to convert the ONNX layers into HLS layers. This paragraph discusses the design challenges involved in converting ONNX layers into another set of FPGA dataflow nodes with the appropriate methods to generate the HLS kernels. The network should contain a set of nodes with an equivalent C++ kernel description to HLS in the FINN hlslibrary. The FINN flow's final stage allows the ONNX operators to infer a C++ description for the corresponding operator from the resulting nodes. Converting each node to a FPGA dataflow Node is how this is done. This wraps around the FINN custom operators class for ONNX and includes several attributes and methods that allow for the inference of the correct HLS description for the kernel. Two things must be considered when converting ONNX layers into HLS layers. The first is to make sure that the graph transformation is correct. This will ensure that HLS layers can be inferred correctly as shown for the Conv operator in Section D.0.5 i.e. the node must be inserted in the right place in the graph and the correct attributes must be inferred Secondly, the FINN hlslibrary should contain an HLS description that matches the ONNX operator.

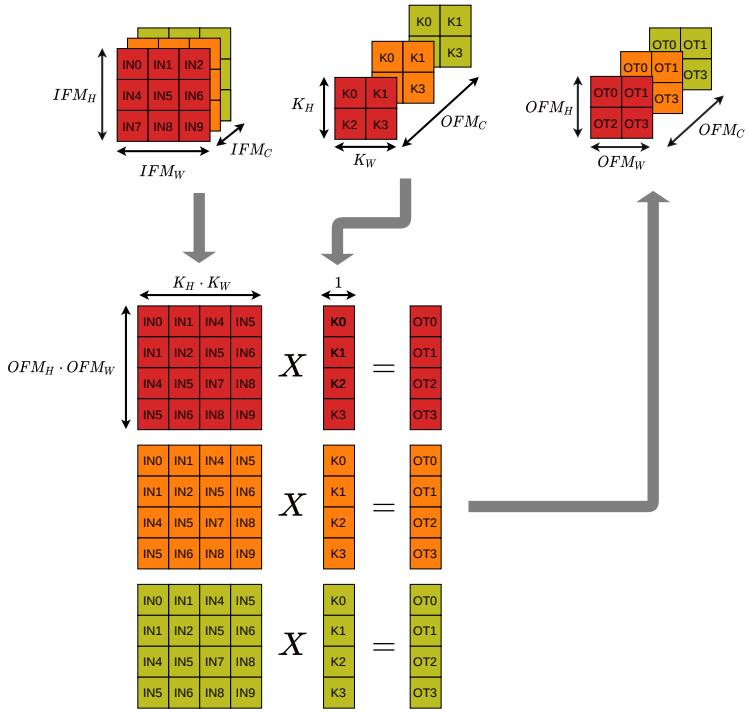


Figure D.3: An example of a convolution lowering method for a depthwise convolution. Note that the image represents the SWU and matrix multiplication from the perspective of the HLS implementation of these kernels. In the ONNX domain of FINN, a depthwise convolution is executed in a similar way as shown in Figure D.2, except that the weight matrix is sparse to ensure that a depthwise convolution is performed taken from the works of [7]

D.0.7. LAYER FOLDING

The FINN-generated DF architecture includes the implementation of loop parallelism and task parallelism, as well as the ability to set data parallelism. Two parameters are required for FINN-HLS layers representations: PE and SIMD [7]. These parameters specify the number of processing elements (PE) within each compute unit (layer), and the number of Single Instruction, Multiple Data, (SIMD), lanes per PE, do note that this type of parallelism only aplies to so called MVAU operator used for convolution layer within FINN, Different types of layers may have different levels or parallelism. These parallelism factors within the FINN are called the folding factors. Only the kernel parallelism is available within convolutional layers. However, the experimental branch has pixel parallelism. This is not covered in the thesis.

The user can also adjust the number of SIMD lanes and PEs in each Layer. This only applies to the so-called MVAU. Different types of layers may have different levels or parallelism.

It is not easy to set the right level of parallelism for each layer. The parallelism factors can be used to adjust two parameters: interference and resource usage. The goal of a parallelism factor is to optimize the interference while preserving the resource limits of

the target device. A designer would aim to adjust the parallelism to match the latency of each layer (compute unit).

The goal of this work is to consume as little power as possible and fit within the Ultra96V2 with a 2 second interference. To achieve the interference goals, it is recommended to choose a low clocking speed with a slight degree of parallelism.

Note that the folding factors of a layer have a linear relationship with the latency of that layer, increasing/decreasing the parallelism by a factor of 2 will decrease/increase the latency by a factor of 2 respectively. The logic and utilization of memory resources will be affected if the layer's parallelism is adjusted. Consider the node that performs matrix multiplication in a convolution with a lower frequency, known as the MVAU. Figure D.4 shows such a matrix multiplication. The first operand is the matrix with weights, and the second operand the lowered image. This Figure is identical to Figure x but with both matrices being transposed and switched in order. The blue color denotes the dimension in which SIMD parallelism is operating, while the green color denotes the PE parallelism. Finally, the red color denotes the MMV parallelism, which is always 1, as explained previously. This layer performs a matrix multiplication between the input pixels, weight matrix of form [MH, MW], where the number of input channels and output channels are respectively. The parallelism factors PE and SIMD are directly related to the number of multipliers and adders used to unroll the matrix multiplication. The expectation is that increasing the parallelism will result in an increase in the logic resources (i.e. DSPs and LUTs will vary depending on the logic resources used to implement the arithmetic). The shape of the array that holds the weights will be determined by the parallelism factors PE or SIMD. The total weight array is composed of MW·MH· W_B bits. W_B bits refering to the number of bits that represent a weight value. Furthermore, note that each PE will have SIMD number of multiplication/additions in parallel between the input pixels and weights.

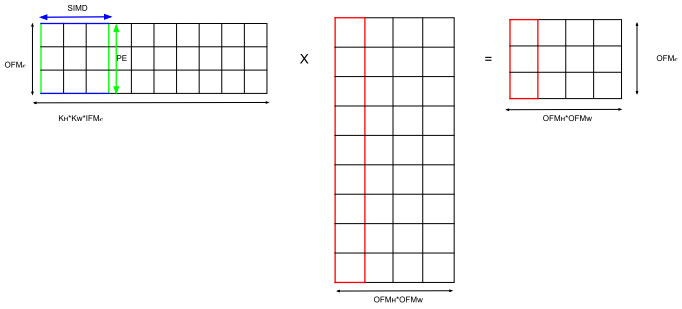


Figure D.4: Convolution operation expressed as matrix multiplication between the weights (first operand) and input image inspired by [7]

Therefore, when the MVAU reads in a new data packet for processing, it should also read PE· SIMD · $W_B$bits from the weight array. Experiments have shown that this results in a weight array of width and depth as expressed by formula X and formula X respectively

$$\text{Width} = PE \cdot \text{SIMD} \cdot W_B$$

$$\text{Depth} = \frac{MW \cdot MH}{PE \cdot SIMD}$$

As certain resources have a fixed shape, e.g. BRAM within the Ultrascale environment can store 36 kbits; either as a single 36 kbits RAM module or two independent 18 kbits RAM modules with a fixed amount of write ports influencing how arrays are partitioned. For example, the width of the weight array is not utilizing the full width of the hardware memory component, the memory component will not be fully occupied and the remaining bits along the width are wasted. Especially for less flexible memory components, such as BRAM, this might lead to severe underutilisation of the memory component. Within FINN analysis exist on whether the BRAM Resources are utilized efficiently which gives a layer by layer output regarding Resource usage. The BRAM efficiency is calculated as formula X

$$BRAMeff = \frac{Width}{BRAM16estcapacity}$$

With $BRAM16estcapacity$ calculated within formula X

$$BRAM16estcapacity = BRAM16_{est} * 36 * 512$$

With $BRAM16_{est}$ being calculated as formula X

$$\text{BRAM16est}(Width, Depth) \triangleq \begin{cases} \left\lceil \frac{Depth}{16384} \right\rceil, & \text{if } Width = 1 \\ \left\lceil \frac{Depth}{8192} \right\rceil, & \text{if } Width = 2 \\ \left\lceil \frac{Depth}{4096} \right\rceil \left\lceil \frac{Width}{4} \right\rceil, & \text{if } Width \leq 4 \\ \left\lceil \frac{Depth}{2048} \right\rceil \left\lceil \frac{Width}{9} \right\rceil, & \text{if } Width \leq 9 \\ \left\lceil \frac{Depth}{1024} \right\rceil \left\lceil \frac{Width}{18} \right\rceil, & \text{if } (Width \leq 18) \vee \left(Depth > 512\right) \\ \left\lceil \frac{Depth}{512} \right\rceil \left\lceil \frac{Width}{36} \right\rceil, & \text{otherwise} \end{cases} \tag{D.4}$$

The wish for the BRAM efficiency to at least be higher than 50% for every layer inside of VLV, as this means that the BRAM usage is nonoptimized allowing the degree of parallelization to be doubled for that particular layer without using more BRAM resources.One of the future features in the pipeline for FINN is automated resource-aware folding.

D.0.8. SELECTING MEMORY MODE

Within FINN the ability is set for individual layers to use different types of primitives for weight memory. The goal of any FPGA design is to have a balanced design in which every primitive is utilized to roughly the same degree. Using the *resType* within a FPGA dataflow node attribute primitive types can be selected.
FINN supports two types of the *mem_mode* attribute for the HLS implementation of activation layers. This mode controls how the weight values are accessed during the execution. Currently, two settings for the mem_mode are supported in FINN:

1. const
2. decoupled

CONST

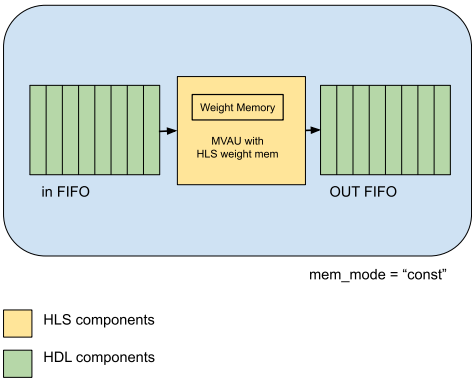


Figure D.5: Figure depicting the architecture of a MVAU FPGA dataflow Component in *const* mode, note how the weights are baked into to the component, Figure inspired by [8].

The *const* mode has the weights *baked in*. This means that they are part of the HLS code. The weight values are included in the code during the IP block generation and synthesized with it. The user can find the code that generated the resulting IP in the FINN HLS Library. As shown in Figure D.6, the resulting IP block contains an input stream and an output stream,FIFOs are connected to these.
Const mode offers a significant advantage over traditional decoupled modes as they use fewer resources resulting in less power usage. However, because it allows the user to control the weight memory primitives with less precision, resource allocation problems can arise. Vivado HLS may not always produce the correct synthesis.

DECOUPLED

A different version of the MVAU with three ports is used in *decoupled* mode. The circuit's input and output streams are connected via Verilog FIFOs. A third input is used to stream weights. The user can find the code that generated the IP in the FINN HLS Library. A Verilog weight streamer component retrieves the weight memory from the MVAU and sends them the values via the third FIFO. This FIFOis located in FINN. For the IP block generation this component, the IP block resulting from the synthesis of the HLS code of the streaming MVAU and a FIFO for the weight stream are combined in a Verilog wrapper. The weight values are saved as.dat files, and then stored in the weight memory that the weight streamer can read. Externally, the MVAU in decoupled mode provides the same inputs and outputs as the const mode MVAU.
Decoupled mode has a larger resource footprint due to the additional weight FIFO and weight streamer. However, it gives the user more control over which primitive they choose to use. Because it has lower synthesis times than Const mode, and can load different weights through the .dat file, prototyping environments have more functionality using *decoupled* memory mode.

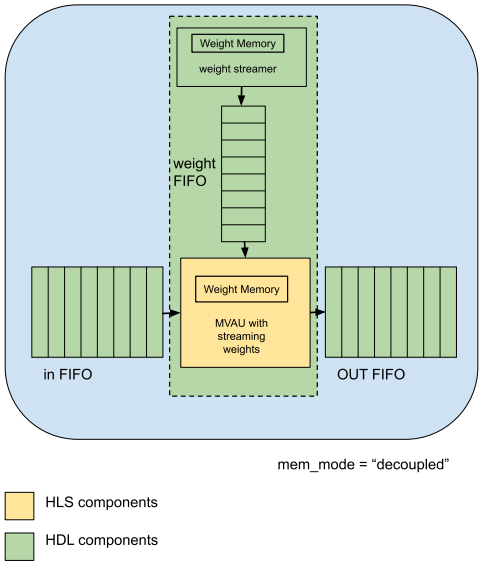


Figure D.6: Figure depicting the architecture of a MVAU FPGA dataflow Component in *Decoupled* mode, note the added weight streamer with the FIFO attached for loading the weights into the weight memory from a .dat file. Figure inspired by [8].

Conclusion

Due to the main objective of the implementation of VLV inside of an FPGA, for the design to be as power and resource efficient as possible as opposed to have a more balanced but heavier resource usage the choice of data type was set at *const*. In an attempt to create a balanced resource usage the final layers of VLV the primitive types have been set to DSP.

D.0.9. Set Fifo depths

Even with all FINN HLS FPGA-dataflow layers appropriately parallelized, it is necessary to insert FIFOs between them to prevent stalls due to burst errors. The sizes of those FIFOs are hard to predict analytically, as such the transform *setFifoDepths* creates very deep FIFO's between all the FPGA-dataflow nodes, creating a stitched together Verilog IP. The design is simulated with a stream of multiple random input images. During the simulation the maximum occupancy for each FIFO is tracked. With the simulation finished the inFIFODepth/outFIFODepth attributes are set to the maximum observed occupancy to a minimum of 0.

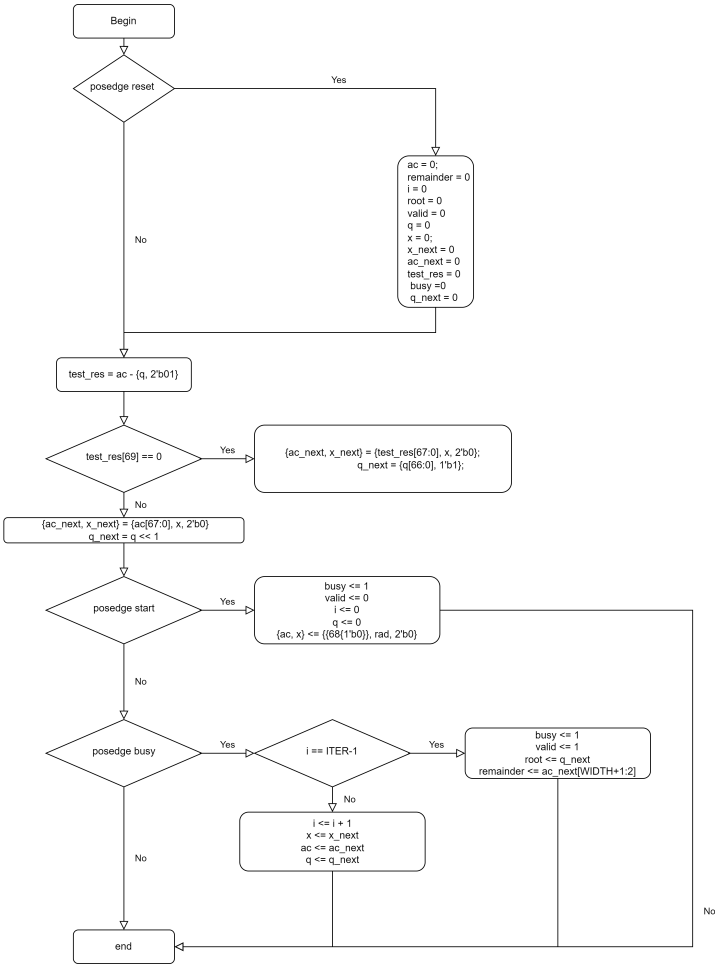
D.0.10. ZYNQ build

Technically with the stitched IP generated the build flow is done and can be implemented within an FPGA. Due to the unique MPSOC ZYNQ attributes off the ultra96V2 an automatic bit file can be generated which can be run via PYNQ within the Arm processor. The *zynqbuild* transform connects the previously generated Stitched IP to an Ultrascale+ MPSOC IP and implements it on the chosen board environment resulting in

a generated bit file and an HWH file to be implanted within the PYNQ environment. The PYNQ transform uses the generated *zynqbuild* transform files and automatically generates a driver to be used inside PYNQ.

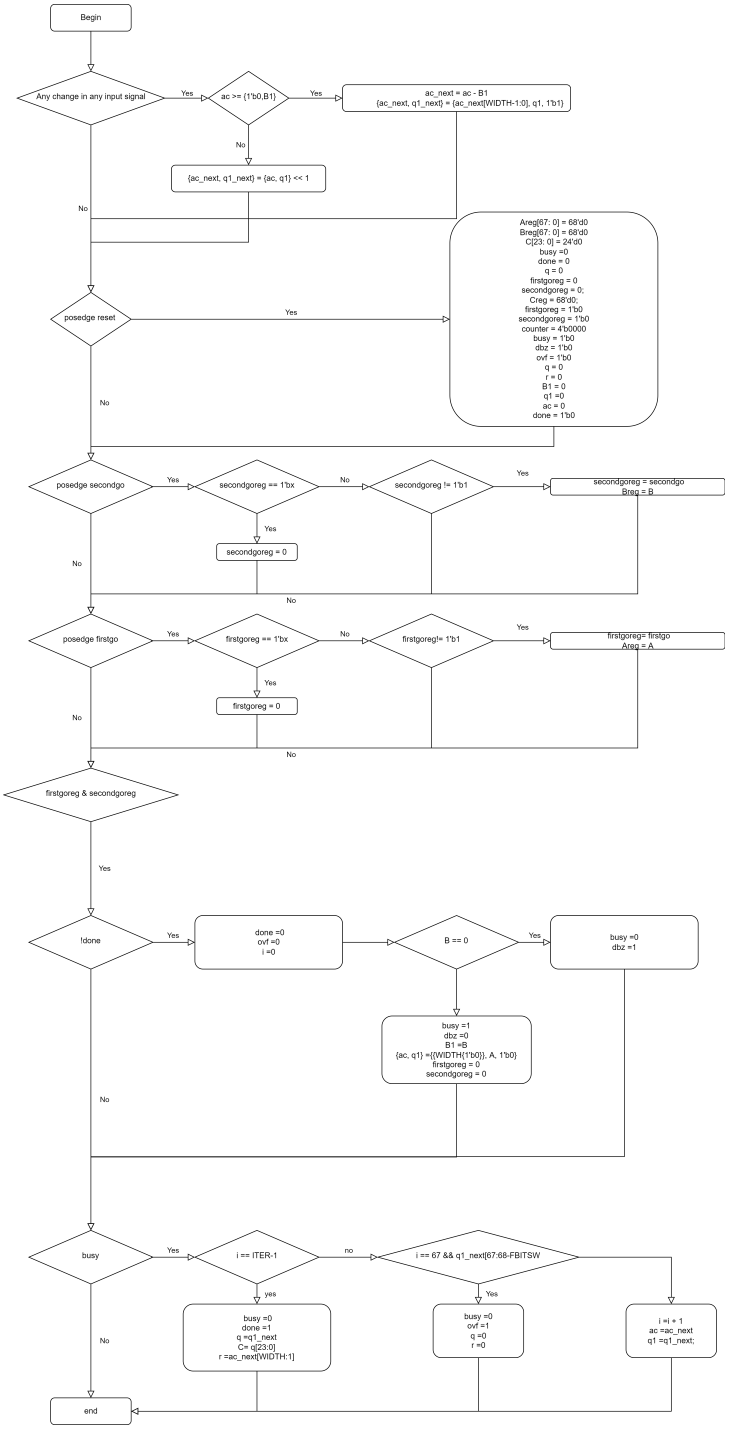
E

SQUARE ROOT DIAGRAM


Begin
posedge reset
Yes
No
ac = 0;
remainder = 0
i = 0
root = 0
valid = 0
q = 0
x = 0;
x_next = 0
ac_next = 0
test_res = 0
busy =0
q_next = 0
test_res = ac - {q, 2'b01}
test_res[69] == 0
Yes
{ac_next, x_next} = {test_res[67:0], x, 2'b0};
q_next = {q[66:0], 1'b1};
No
{ac_next, x_next} = {ac[67:0], x, 2'b0}
q_next = q << 1
posedge start
Yes
busy <= 1
valid <= 0
i <= 0
q <= 0
{ac, x} <= {{68{1'b0}}, rad, 2'b0}
No
posedge busy
Yes
i == ITER-1
Yes
busy <= 1
valid <= 1
root <= q_next
remainder <= ac_next[WIDTH+1:2]
No
i <= i + 1
x <= x_next
ac <= ac_next
q <= q_next
No
No
end

F

DIVIDER DIAGRAM


Begin
Any change in any input signal
Yes
ac >= {1'b0,B1}
Yes
ac_next = ac - B1
{ac_next, q1_next} = {ac_next[WIDTH-1:0], q1, 1'b1}
No
{ac_next, q1_next} = {ac, q1} << 1
No
Areg[67: 0] = 68'd0
Breg[67: 0] = 68'd0
C[23: 0] = 24'd0
busy =0
done = 0
q = 0
firstgoreg = 0
secondgoreg = 0;
Creg = 68'd0;
firstgoreg = 1'b0
secondgoreg = 1'b0
counter = 4'b0000
busy = 1'b0
dbz = 1'b0
ovf = 1'b0
q = 0
r = 0
B1 = 0
q1 =0
ac = 0
done = 1'b0
posedge reset
Yes
No
posedge secondgo
Yes
secondgoreg == 1'bx
No
secondgoreg != 1'b1
Yes
secondgoreg = secondgo
Breg = B
Yes
secondgoreg = 0
No
No
posedge firstgo
Yes
firstgoreg == 1'bx
No
firstgoreg!= 1'b1
Yes
firstgoreg= firstgo
Areg = A
Yes
firstgoreg = 0
No
No
firstgoreg & secondgoreg
Yes
!done
Yes
done =0
ovf =0
i =0
B == 0
Yes
busy =0
dbz =1
busy =1
dbz =0
B1 =B
{ac, q1} ={{WIDTH{1'b0}}, A, 1'b0}
firstgoreg = 0
secondgoreg = 0
No
busy
Yes
i == ITER-1
no
i == 67 && q1_next[67:68-FBITSW
yes
busy =0
done =1
q =q1_next
C= q[23:0]
r =ac_next[WIDTH:1]
Yes
busy =0
ovf =1
q =0
r =0
i =i + 1
ac =ac_next
q1 =q1_next;
No
end

G

NNgen Log

NNgenlog.txt

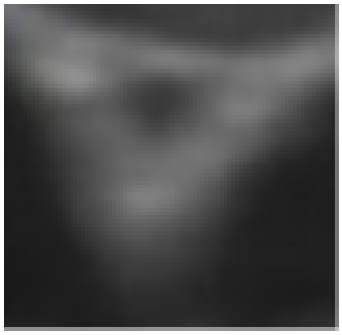
```
/home/christiaan/nngen_test/python3/bin/python3 "/home/christiaan/nngen/examples/RCNN Example/make.py"
NNgen: Neural Network Accelerator Generator version 1.3.0
[IP-XACT]
  Output: RCNN
[Configuration]
AXI Master Interface
  Data width   : 32
  Address width: 32
AXI Slave Interface
  Data width   : 32
  Address width: 32
[Schedule Table]
Stage 0
Stage 1
  <conv2d Conv_0 dtype:int16 shape:1, 448, 448, 16 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:16, scale:1, cshamt_out:30 act_func:relu sum_dtype:int64 concur_och:2
   <placeholder act dtype:int16 shape:1, 448, 448, 3 default_addr:1280 g_index:2 word_alignment:2 aligned_shape:1, 448, 448, 4 layout:'N', 'H', 'W', 'C' onnx_layout
   <variable 97 dtype:int16 shape:16, 3, 3, 3 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:16, 3, 3, 4 layout:'O', 'H', 'W', 'I' onnx_layout:'O', '
   <variable 98 dtype:int32 shape:16, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:16, scale_factor:594081.944199>
   <variable onnx_Conv_0_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 2
  <max_pool_serial MaxPool_2 dtype:int16 shape:1, 224, 224, 16 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:41654016 g_index:0 l_ind
   <conv2d Conv_0 dtype:int16 shape:1, 448, 448, 16 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:16, scale:1, cshamt_out:30 act_func:relu sum_dtype:int64 concur_och:2
Stage 3
  <conv2d Conv_3 dtype:int16 shape:1, 224, 224, 32 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:32, scale:1, cshamt_out:30 act_func:relu sum_dtype:int64 concur_och:4
   <max_pool_serial MaxPool_2 dtype:int16 shape:1, 224, 224, 16 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:41654016 g_index:0 l_in
   <variable 100 dtype:int16 shape:32, 3, 3, 16 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:32, 3, 3, 16 layout:'O', 'H', 'W', 'I' onnx_layout:'O'
   <variable 101 dtype:int32 shape:32, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:32, scale_factor:809352.345566>
   <variable onnx_Conv_3_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 4
  <max_pool_serial MaxPool_5 dtype:int16 shape:1, 112, 112, 32 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:46470912 g_index:0 l_ind
   <conv2d Conv_3 dtype:int16 shape:1, 224, 224, 32 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:32, scale:1, cshamt_out:30 act_func:relu sum_dtype:int64 concur_och:4
Stage 5
  <conv2d Conv_6 dtype:int16 shape:1, 112, 112, 64 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:64, scale:1, cshamt_out:31 act_func:relu sum_dtype:int64 concur_och:8
   <max_pool_serial MaxPool_5 dtype:int16 shape:1, 112, 112, 32 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:46470912 g_index:0 l_in
   <variable 103 dtype:int16 shape:64, 3, 3, 32 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:64, 3, 3, 32 layout:'O', 'H', 'W', 'I' onnx_layout:'O'
   <variable 104 dtype:int32 shape:64, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:64, scale_factor:2924308.372883>
   <variable onnx_Conv_6_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 6
  <max_pool_serial MaxPool_8 dtype:int16 shape:1, 56, 56, 64 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:48879360 g_index:0 l_index
   <conv2d Conv_6 dtype:int16 shape:1, 112, 112, 64 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:64, scale:1, cshamt_out:31 act_func:relu sum_dtype:int64 concur_och:8
Stage 7
  <conv2d Conv_9 dtype:int16 shape:1, 56, 56, 128 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:128, scale:1, cshamt_out:31 act_func:relu sum_dtype:int64 concur_och:18
   <max_pool_serial MaxPool_8 dtype:int16 shape:1, 56, 56, 64 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:48879360 g_index:0 l_inde
   <variable 106 dtype:int16 shape:128, 3, 3, 64 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:128, 3, 3, 64 layout:'O', 'H', 'W', 'I' onnx_layout:'
   <variable 107 dtype:int32 shape:128, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:128, scale_factor:9378591.204066>
   <variable onnx_Conv_9_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 8
  <max_pool_serial MaxPool_11 dtype:int16 shape:1, 28, 28, 128 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:50083584 g_index:0 l_ind
   <conv2d Conv_9 dtype:int16 shape:1, 56, 56, 128 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:128, scale:1, cshamt_out:31 act_func:relu sum_dtype:int64 concur_och:1
Stage 9
  <conv2d Conv_12 dtype:int16 shape:1, 28, 28, 256 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:256, scale:1, cshamt_out:33 act_func:relu sum_dtype:int64 concur_och:1
   <max_pool_serial MaxPool_11 dtype:int16 shape:1, 28, 28, 128 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:50083584 g_index:0 l_in
   <variable 109 dtype:int16 shape:256, 3, 3, 128 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:256, 3, 3, 128 layout:'O', 'H', 'W', 'I' onnx_layout
   <variable 110 dtype:int32 shape:256, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:256, scale_factor:48563589.857899>
   <variable onnx_Conv_12_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
```

```
Stage 10
  <max_pool_serial MaxPool_14 dtype:int16 shape:1, 14, 14, 256 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:50685696 g_inde
   <conv2d Conv_12 dtype:int16 shape:1, 28, 28, 256 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:256, scale:1, cshamt_out:33 act_func:relu sum_dtype:int64 co
Stage 11
  <conv2d Conv_15 dtype:int16 shape:1, 14, 14, 512 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:512, scale:1, cshamt_out:34 act_func:relu sum_dtype:int64 con
   <max_pool_serial MaxPool_14 dtype:int16 shape:1, 14, 14, 256 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:50685696 g_ind
   <variable 112 dtype:int16 shape:512, 3, 3, 256 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:512, 3, 3, 256 layout:'O', 'H', 'W', 'I' on
   <variable 113 dtype:int32 shape:512, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:512, scale_factor:89154998.747469>
   <variable onnx_Conv_15_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 12
  <max_pool_serial MaxPool_17 dtype:int16 shape:1, 7, 7, 512 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:50986752 g_index:
   <conv2d Conv_15 dtype:int16 shape:1, 14, 14, 512 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:512, scale:1, cshamt_out:34 act_func:relu sum_dtype:int64 co
Stage 13
  <conv2d Conv_18 dtype:int16 shape:1, 5, 5, 1024 strides:1, 1, 1, 1 padding:0, 0, 0, 0 bias:1024, scale:1, cshamt_out:34 act_func:relu sum_dtype:int64 con
   <max_pool_serial MaxPool_17 dtype:int16 shape:1, 7, 7, 512 ksize:1, 2, 2, 1 strides:1, 2, 2, 1 padding:0, 0, 0, 0 no_reuse default_addr:50986752 g_index
   <variable 115 dtype:int16 shape:1024, 3, 3, 512 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:1024, 3, 3, 512 layout:'O', 'H', 'W', 'I'
   <variable 116 dtype:int32 shape:1024, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:1024, scale_factor:112183695.392112>
   <variable onnx_Conv_18_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 14
  <conv2d Conv_20 dtype:int16 shape:1, 5, 5, 1024 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:1024, scale:1, cshamt_out:35 act_func:relu sum_dtype:int64 con
   <conv2d Conv_18 dtype:int16 shape:1, 5, 5, 1024 strides:1, 1, 1, 1 padding:0, 0, 0, 0 bias:1024, scale:1, cshamt_out:34 act_func:relu sum_dtype:int64 co
   <variable 118 dtype:int16 shape:1024, 3, 3, 1024 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:1024, 3, 3, 1024 layout:'O', 'H', 'W', 'I
   <variable 119 dtype:int32 shape:1024, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:1024, scale_factor:150058892.051316>
   <variable onnx_Conv_20_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 15
  <conv2d Conv_22 dtype:int16 shape:1, 7, 7, 30 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:30, scale:1, cshamt_out:32 act_func:relu sum_dtype:int64 concur_
   <conv2d Conv_20 dtype:int16 shape:1, 5, 5, 1024 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:1024, scale:1, cshamt_out:35 act_func:relu sum_dtype:int64 co
   <variable 121 dtype:int16 shape:30, 1, 1, 1024 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:30, 1, 1, 1024 layout:'O', 'H', 'W', 'I' on
   <variable 122 dtype:int32 shape:30, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:30, scale_factor:40511112.926941>
   <variable onnx_Conv_22_conv.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 16
  <_lazy_reshape Flatten_24 dtype:int16 shape:1, 1470 alias_of:Conv_22 default_addr:51139328 g_index:0 l_index:15 word_alignment:2 aligned_shape:1, 1470 sc
   <conv2d Conv_22 dtype:int16 shape:1, 7, 7, 30 strides:1, 1, 1, 1 padding:1, 1, 1, 1 bias:30, scale:1, cshamt_out:32 act_func:relu sum_dtype:int64 concur
Stage 17
  <matmul Gemm_25 dtype:int16 shape:1, 496 bias:496, scale:1, cshamt_out:33 act_func:relu sum_dtype:int64 concur_out_col:2 stationary:right keep_left defau
   <_lazy_reshape Flatten_24 dtype:int16 shape:1, 1470 alias_of:Conv_22 default_addr:51139328 g_index:0 l_index:15 word_alignment:2 aligned_shape:1, 1470 s
   <variable fcs.1.weight dtype:int16 shape:496, 1470 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:496, 1470 scale_factor:1192254.230479>
   <variable fcs.1.bias dtype:int32 shape:496, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:496, scale_factor:368485044.734294>
   <variable onnx_Gemm_25_gemm.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
Stage 18
  <matmul Gemm_27 dtype:int16 shape:1, 637 bias:637, scale:1, cshamt_out:33 sum_dtype:int64 concur_out_col:8 stationary:right keep_left default_addr:0 g_in
   <matmul Gemm_25 dtype:int16 shape:1, 496 bias:496, scale:1, cshamt_out:33 act_func:relu sum_dtype:int64 concur_out_col:2 stationary:right keep_left defa
   <variable fcs.4.weight dtype:int16 shape:637, 496 default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:637, 496 scale_factor:701433.668319>
   <variable fcs.4.bias dtype:int32 shape:637, default_addr:1606912 g_index:3 word_alignment:1 aligned_shape:637, scale_factor:985946383.808247>
   <variable onnx_Gemm_27_gemm.scale dtype:int16 shape:1, default_addr:1606912 g_index:3 word_alignment:2 aligned_shape:2, scale_factor:32767.000000>
[RAM spec: num]
  32-bit 1024-entry 2-port 1-bank RAM: 1
  16-bit 32768-entry 2-port 2-bank RAM: 1
  16-bit 8192-entry 2-port 2-bank RAM: 1
  16-bit 4096-entry 2-port 2-bank RAM: 18
  16-bit 2048-entry 2-port 2-bank RAM: 1
  16-bit 512-entry 2-port 2-bank RAM: 1
[Substream spec: num]
  'acc_rshift_round_frac', 64, 0, True, 64, 0, True: 1
  'add_tree', 64, 0, True, 1: 1
  'add_tree', 64, 0, True, 9: 1
  'mul_rshift_clip', 64, 0, True, 16, 0, True, 80, 0, True, 16, 0, True: 1
  'mul_rshift_round_madd', 16, 0, True, 16, 0, True, 32, 0, True: 9
  'reduce_max', 16, 0, True: 1
[Stream spec: num]
  <class 'nngen.operator.conv2d.conv2d'>, <dtype int16>, <dtype int16>, <dtype int32>, <dtype int16>, <dtype int16>, 1, 3, 3, None, <dtype int64>, 1, 1, 1,
  <class 'nngen.operator.pool_serial.max_pool_serial'>, <dtype int16>, <dtype int16>, 1, 2, 2, True, 1: 1
  <class 'nngen.operator.conv2d.conv2d'>, <dtype int16>, <dtype int16>, <dtype int32>, <dtype int16>, <dtype int16>, 1, 1, 1, None, <dtype int64>, 1, 1, 1,
  <class 'nngen.operator.basic._lazy_reshape'>, <dtype int16>, <dtype int16>, 1, True: 1
  <class 'nngen.operator.matmul.matmul'>, <dtype int16>, <dtype int16>, <dtype int32>, <dtype int16>, <dtype int16>, 1, 1, 1, None, <dtype int64>, 1, 1, 1,
[Control name # states: num]
  main_fsm # states: 163
  control_conv2d_24 # states: 56
  control_max_pool_serial_26 # states: 26
  control_conv2d_54 # states: 40
  control_matmul_58 # states: 40
[Register Map]
   0 R : header0 default: 0
   4 R : header1 default: 0
   8 R : header2 default: 0
  12 R : header3 default: 0
  16  W: Start set '1' to run
  20 R : Busy returns '1' when running
  24  W: Reset set '1' to initialize internal logic
  28 R : Opcode from extern objects to SW returns '0' when idle
  32  W: Resume extern objects set '1' to resume
  36 RW: Global address offset default: 0
  40 RW: Address of temporal storages size: 15539KB
  44 RW: Address of output matmul 'Gemm_27' size: 2KB, dtype: int16, shape: 1, 637, alignment: 2 words 4 bytes, aligned shape: 1, 638
  48 RW: Address of placeholder 'act' size: 1568KB, dtype: int16, shape: 1, 448, 448, 3, alignment: 2 words 4 bytes, aligned shape: 1, 448, 448, 4
  52 RW: Address of variables '97', '98', 'onnx_Conv_0_conv.scale', '100', '101', 'onnx_Conv_3_conv.scale', '103', '104', 'onnx_Conv_6_conv.scale', '106',
[Default Memory Map start - end] entire range: [0 - 51143295], size: 49945KB
  [       0 -     1279]: output matmul 'Gemm_27' size: 2KB, dtype: int16, shape: 1, 637, alignment: 2 words 4 bytes, aligned shape: 1, 638
  [    1280 -  1606911]: placeholder 'act' size: 1568KB, dtype: int16, shape: 1, 448, 448, 3, alignment: 2 words 4 bytes, aligned shape: 1, 448, 448, 4
  [ 1606912 -  1608063]: variable '97' size: 2KB, dtype: int16, shape: 16, 3, 3, 3, alignment: 2 words 4 bytes, aligned shape: 16, 3, 3, 4
```

```
[ 1608064 -  1608127]: variable '98' size: 64B, dtype: int32, shape: 16,, alignment: 1 words 4 bytes, aligned shape: 16,
[ 1608128 -  1608191]: variable 'onnx_Conv_0_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[ 1608192 -  1617407]: variable '100' size: 9KB, dtype: int16, shape: 32, 3, 3, 16, alignment: 2 words 4 bytes, aligned shape: 32, 3, 3, 16
[ 1617408 -  1617535]: variable '101' size: 128B, dtype: int32, shape: 32,, alignment: 1 words 4 bytes, aligned shape: 32,
[ 1617536 -  1617599]: variable 'onnx_Conv_3_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[ 1617600 -  1654463]: variable '103' size: 36KB, dtype: int16, shape: 64, 3, 3, 32, alignment: 2 words 4 bytes, aligned shape: 64, 3, 3, 32
[ 1654464 -  1654719]: variable '104' size: 256B, dtype: int32, shape: 64,, alignment: 1 words 4 bytes, aligned shape: 64,
[ 1654720 -  1654783]: variable 'onnx_Conv_6_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[ 1654784 -  1802239]: variable '106' size: 144KB, dtype: int16, shape: 128, 3, 3, 64, alignment: 2 words 4 bytes, aligned shape: 128, 3, 3, 64
[ 1802240 -  1802751]: variable '107' size: 512B, dtype: int32, shape: 128,, alignment: 1 words 4 bytes, aligned shape: 128,
[ 1802752 -  1802815]: variable 'onnx_Conv_9_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[ 1802816 -  2392639]: variable '109' size: 576KB, dtype: int16, shape: 256, 3, 3, 128, alignment: 2 words 4 bytes, aligned shape: 256, 3, 3, 128
[ 2392640 -  2393663]: variable '110' size: 1KB, dtype: int32, shape: 256,, alignment: 1 words 4 bytes, aligned shape: 256,
[ 2393664 -  2393727]: variable 'onnx_Conv_12_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[ 2393728 -  4753023]: variable '112' size: 2304KB, dtype: int16, shape: 512, 3, 3, 256, alignment: 2 words 4 bytes, aligned shape: 512, 3, 3, 256
[ 4753024 -  4755071]: variable '113' size: 2KB, dtype: int32, shape: 512,, alignment: 1 words 4 bytes, aligned shape: 512,
[ 4755072 -  4755135]: variable 'onnx_Conv_15_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[ 4755136 - 14192319]: variable '115' size: 9216KB, dtype: int16, shape: 1024, 3, 3, 512, alignment: 2 words 4 bytes, aligned shape: 1024, 3, 3, 512
[14192320 - 14196415]: variable '116' size: 4KB, dtype: int32, shape: 1024,, alignment: 1 words 4 bytes, aligned shape: 1024,
[14196416 - 14196479]: variable 'onnx_Conv_18_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[14196480 - 33070847]: variable '118' size: 18432KB, dtype: int16, shape: 1024, 3, 3, 1024, alignment: 2 words 4 bytes, aligned shape: 1024, 3, 3, 1024
[33070848 - 33074943]: variable '119' size: 4KB, dtype: int32, shape: 1024,, alignment: 1 words 4 bytes, aligned shape: 1024,
[33074944 - 33075007]: variable 'onnx_Conv_20_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[33075008 - 33136447]: variable '121' size: 60KB, dtype: int16, shape: 30, 1, 1, 1024, alignment: 2 words 4 bytes, aligned shape: 30, 1, 1, 1024
[33136448 - 33136575]: variable '122' size: 128B, dtype: int32, shape: 30,, alignment: 1 words 4 bytes, aligned shape: 30,
[33136576 - 33136639]: variable 'onnx_Conv_22_conv.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[33136640 - 34594879]: variable 'fcs.1.weight' size: 1425KB, dtype: int16, shape: 496, 1470, alignment: 2 words 4 bytes, aligned shape: 496, 1470
[34594880 - 34596863]: variable 'fcs.1.bias' size: 2KB, dtype: int32, shape: 496,, alignment: 1 words 4 bytes, aligned shape: 496,
[34596864 - 34596927]: variable 'onnx_Gemm_25_gemm.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[34596928 - 35228863]: variable 'fcs.4.weight' size: 618KB, dtype: int16, shape: 637, 496, alignment: 2 words 4 bytes, aligned shape: 637, 496
[35228864 - 35231423]: variable 'fcs.4.bias' size: 3KB, dtype: int32, shape: 637,, alignment: 1 words 4 bytes, aligned shape: 637,
[35231424 - 35231487]: variable 'onnx_Gemm_27_gemm.scale' size: 64B, dtype: int16, shape: 1,, alignment: 2 words 4 bytes, aligned shape: 2,
[35231488 - 51143295]: temporal storages size: 15539KB
```

H

TEMPLATE USED

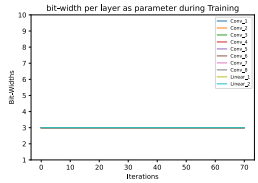


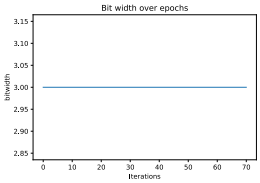
I

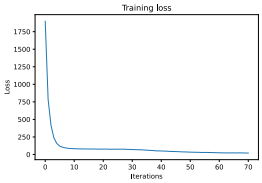
FULL RESULTS APPENDIX

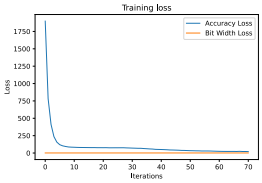
I.1. 2 BITS FULL RESULTS

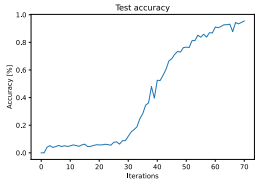
I.1.1. POWER 0


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2


Bit width over epochs
bitwidth
Iterations

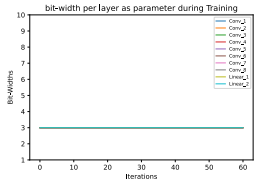

Training loss
Loss
Iterations

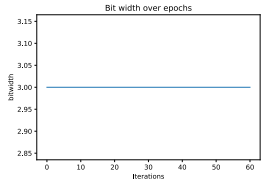

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

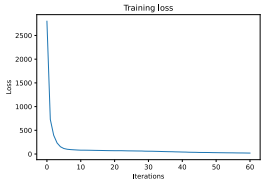

Test accuracy
Accuracy (%)
Iterations

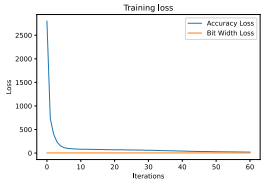
I

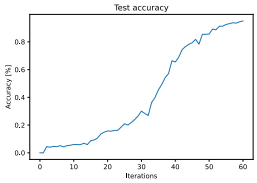
I.1.2. POWER 1


bit-width per layer as parameter during Training
Bit-Widths
Iterations

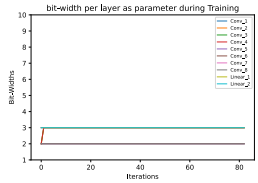

Bit width over epochs
bitwidth
Iterations

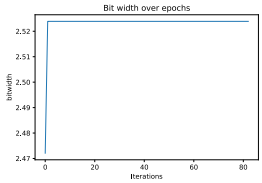

Training loss
Loss
Iterations

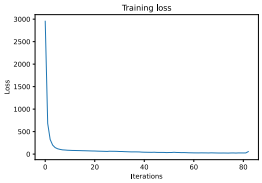

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

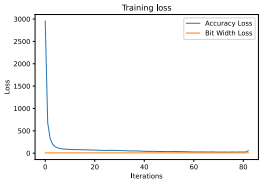

Test accuracy
Accuracy (%)
Iterations

I.1.3. POWER 2


bit-width per layer as parameter during Training
Bit-Widths
Iterations

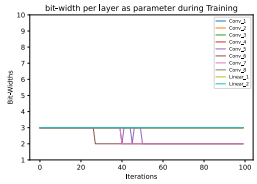

Bit width over epochs
bitwidth
Iterations

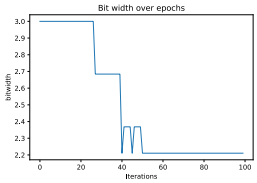

Training loss
Loss
Iterations

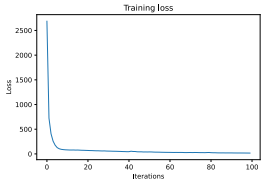

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

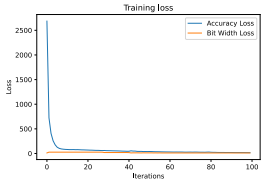

Test accuracy
Accuracy [%]
Iterations

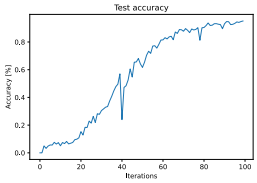
I.1.4. POWER 3


bit-width per layer as parameter during Training
Bit-Widths
Iterations

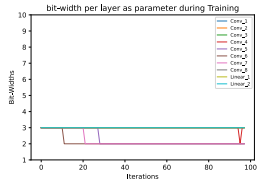

Bit width over epochs
bitwidth
Iterations

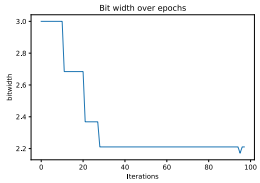

Training loss
Loss
Iterations

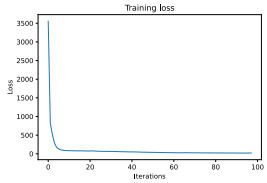

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

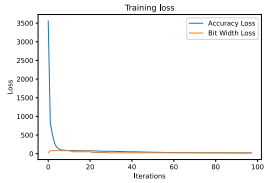

Test accuracy
Accuracy (%)
Iterations

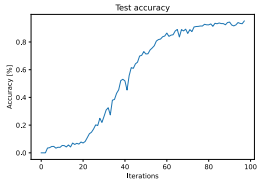
I.1.5. POWER 4


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2

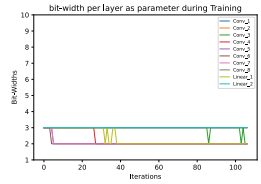

Bit width over epochs
bitwidth
Iterations

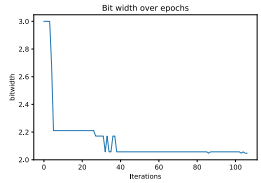

Training loss
Loss
Iterations

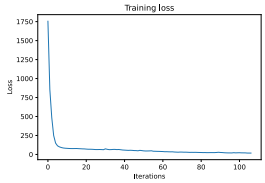

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

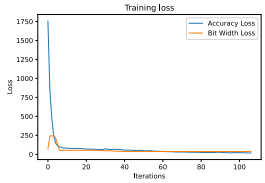

Test accuracy
Accuracy [%]
Iterations

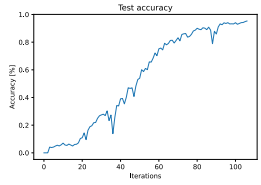
I.1.6. POWER 5


bit-width per layer as parameter during Training
Bit-Widths
Iterations

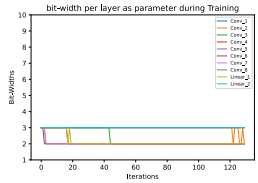

Bit width over epochs
bitwidth
Iterations

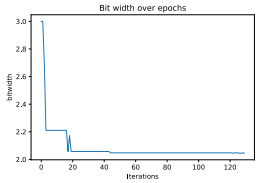

Training loss
Loss
Iterations

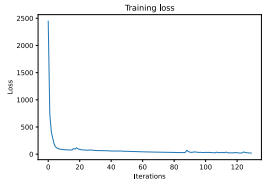

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

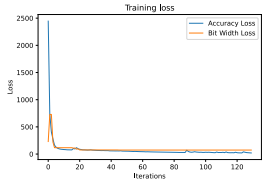

Test accuracy
Accuracy (%)
Iterations

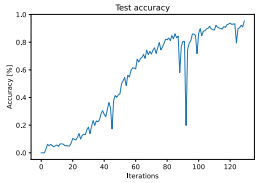
I.1.7. POWER 6


bit-width per layer as parameter during Training
Bit-Widths
Iterations


Bit width over epochs
bitwidth
Iterations

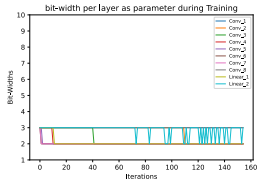

Training loss
Loss
Iterations

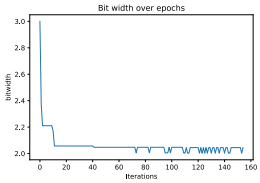

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

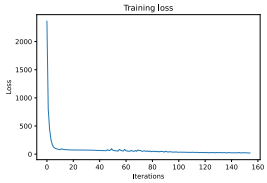

Test accuracy
Accuracy (%)
Iterations

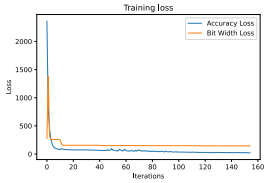
I

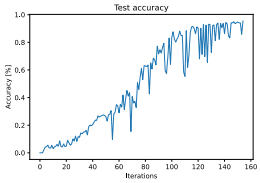
I.1.8. POWER 7


bit-width per layer as parameter during Training
Bit-Widths
Iterations

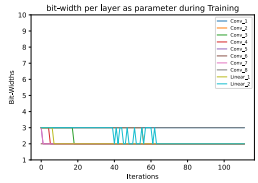

Bit width over epochs
bitwidth
Iterations

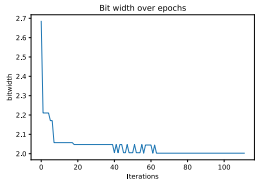

Training loss
Loss
Iterations

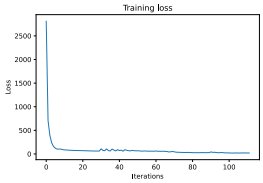

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

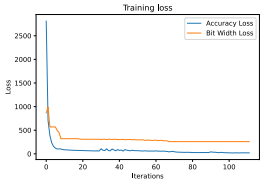

Test accuracy
Accuracy (%)
Iterations

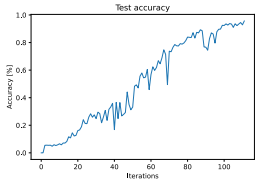
I.1.9. POWER 8


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2


Bit width over epochs
bitwidth
Iterations

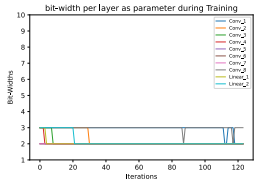

Training loss
Loss
Iterations

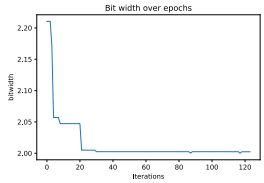

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

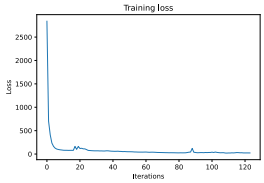

Test accuracy
Accuracy (%)
Iterations

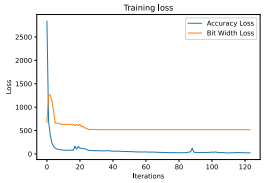
I

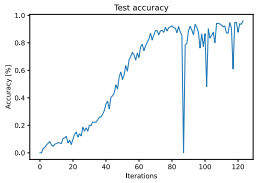
I.1.10. POWER 9


bit-width per layer as parameter during Training
Bit-Widths
Iterations

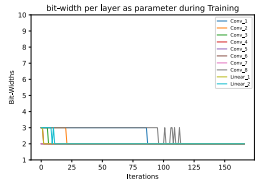

Bit width over epochs
Bitwidth
Iterations

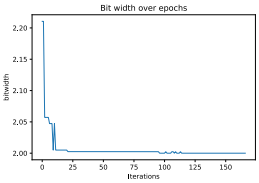

Training loss
Loss
Iterations

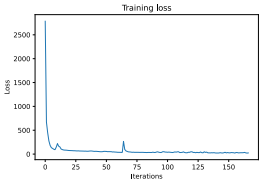

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

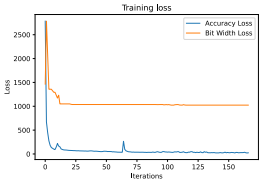

Test accuracy
Accuracy (%)
Iterations

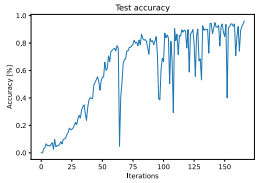
I.1.11. POWER 10





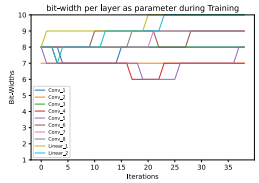


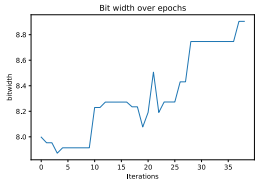


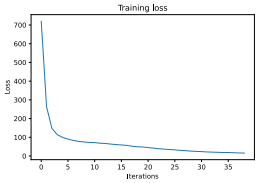


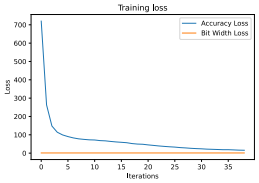
I.2. 8 BITS FULL RESULTS

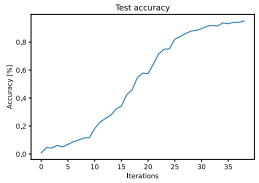
I.2.1. POWER 0


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2


Bit width over epochs
bitwidth
Iterations

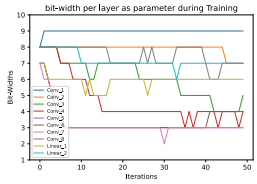

Training loss
Loss
Iterations

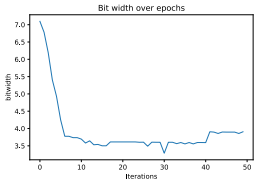

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

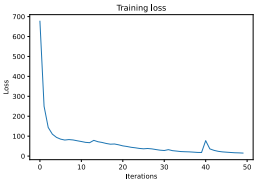

Test accuracy
Accuracy (%)
Iterations

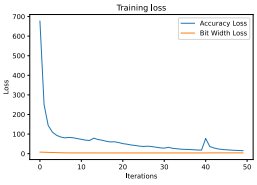
I

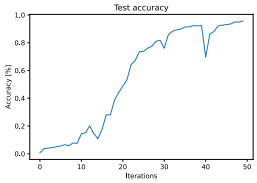
I.2.2. POWER 1


bit-width per layer as parameter during Training
Bit-Widths
Iterations

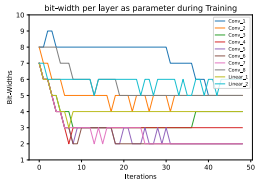

Bit width over epochs
bitwidth
Iterations

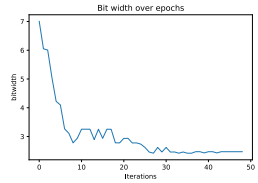

Training loss
Loss
Iterations

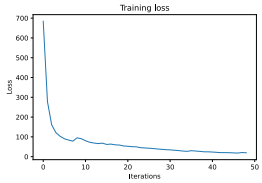

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

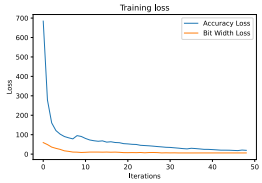

Test accuracy
Accuracy (%)
Iterations

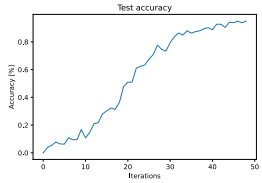
I.2.3. POWER 2


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2


Bit width over epochs
bitwidth
Iterations

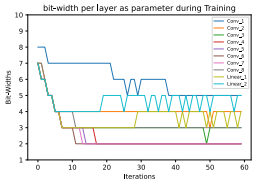

Training loss
Loss
Iterations

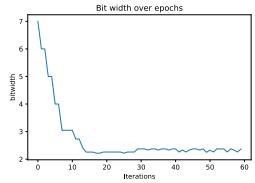

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

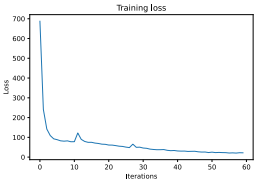

Test accuracy
Accuracy (%)
Iterations

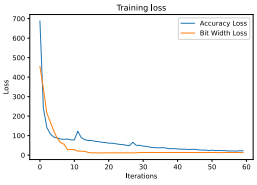
I

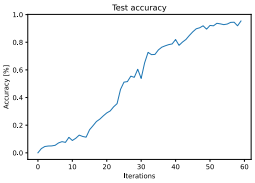
I.2.4. POWER 3


bit-width per layer as parameter during Training
Bit-Widths
Iterations

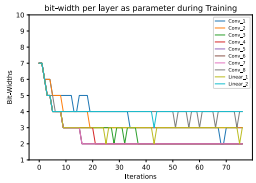

Bit width over epochs
bitwidth
Iterations

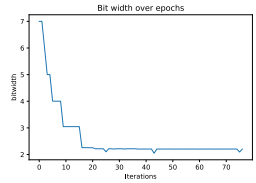

Training loss
Loss
Iterations

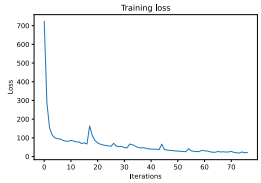

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

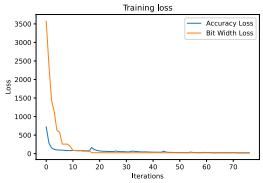

Test accuracy
Accuracy (%)
Iterations

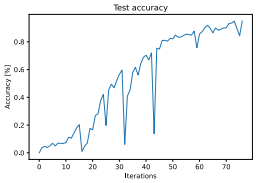
I.2.5. POWER 4


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2


Bit width over epochs
bitwidth
Iterations

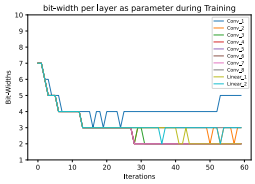

Training loss
Loss
Iterations

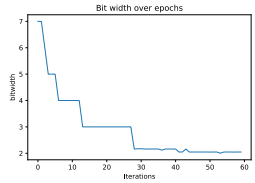

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

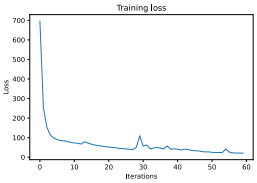

Test accuracy
Accuracy (%)
Iterations

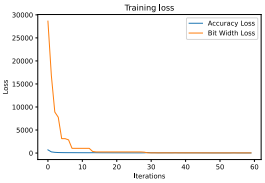
I

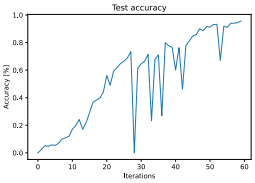
I.2.6. POWER 5


bit-width per layer as parameter during Training
Bit-Widths
Iterations

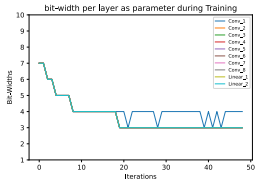

Bit width over epochs
bitwidth
Iterations

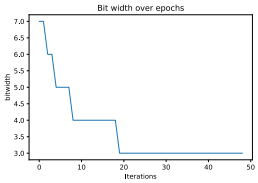

Training loss
Loss
Iterations

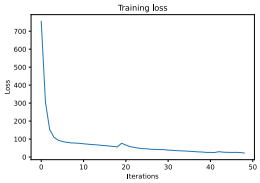

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

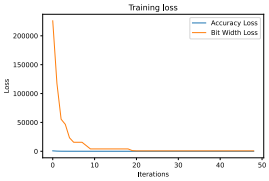

Test accuracy
Accuracy (%)
Iterations

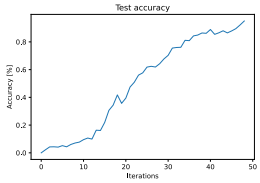
I.2.7. POWER 6


bit-width per layer as parameter during Training
Bit-Widths
Iterations


Bit width over epochs
bitwidth
Iterations

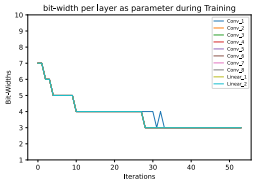

Training loss
Loss
Iterations

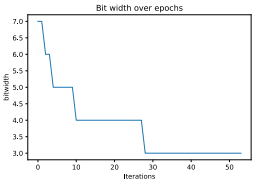

Training loss
Accuracy Loss
Bit Width Loss
Loss
Iterations

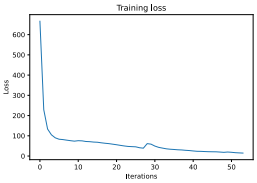

Test accuracy
Accuracy [%]
Iterations

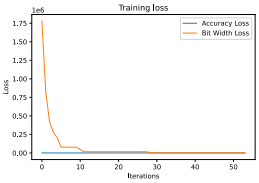
I

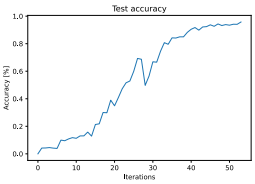
I.2.8. POWER 7


bit-width per layer as parameter during Training
Bit-Widths
Iterations

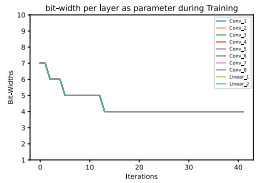

Bit width over epochs
bitwidth
Iterations

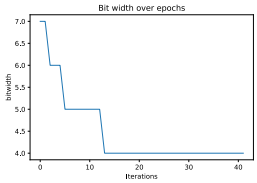

Training loss
Loss
Iterations

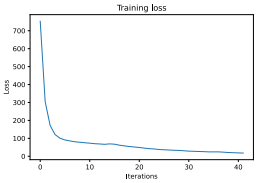

Training loss
1e6
Accuracy Loss
Bit Width Loss
Loss
Iterations

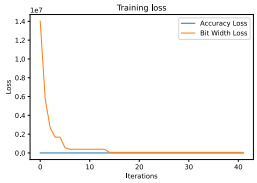

Test accuracy
Accuracy (%)
Iterations

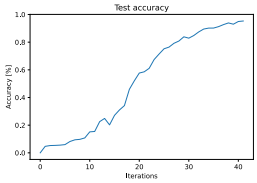
I.2.9. POWER 8


bit-width per layer as parameter during Training
Bit-Widths
Iterations
Conv_1
Conv_2
Conv_3
Conv_4
Conv_5
Conv_6
Conv_7
Conv_8
Linear_1
Linear_2


Bit width over epochs
bitwidth
Iterations

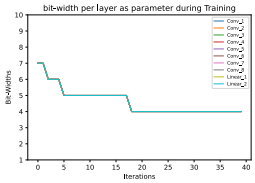

Training loss
Loss
Iterations

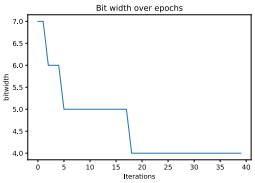

Training loss
1e7
Accuracy Loss
Bit Width Loss
Loss
Iterations

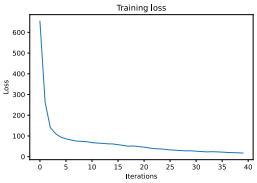

Test accuracy
Accuracy (%)
Iterations

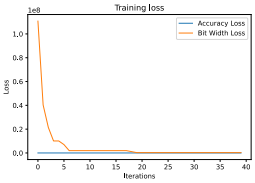
I

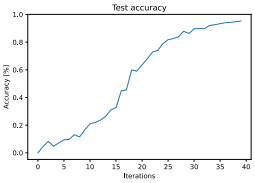
I.2.10. POWER 9


bit-width per layer as parameter during Training
Bit-Widths
Iterations

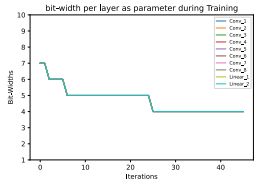

Bit width over epochs
bitwidth
Iterations

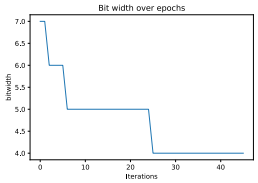

Training loss
Loss
Iterations

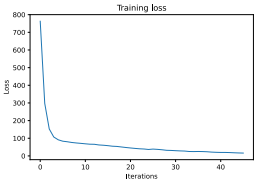

Training loss
1e8
Accuracy Loss
Bit Width Loss
Loss
Iterations

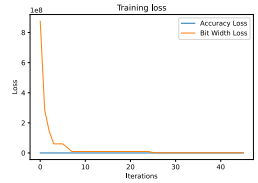

Test accuracy
Accuracy (%)
Iterations

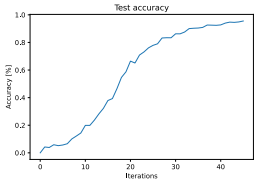
I.2.11. POWER 10











REFERENCES

[1] Tiago Costa, Chen Shi, Kevin Tien, and Kenneth L Shepard. A cmos 2d transmit beamformer with integrated pzt ultrasound transducers for neuromodulation. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE, 2019.

[2] H.e Zhou. Wearable ultrasound improves motor function in an mptp mouse model of parkinson's disease. *IEEE Transactions on Biomedical Engineering*, 66(11):3006–3013.

[3] P.D. Vida Pashaei. Flexible body-conformal ultrasound patches for image-guided neuromodulation.

[4] Akinori Inamura, Sadahiro Nomura, Hirokazu Sadahiro, Hirochika Imoto, Hideyuki Ishihara, and Michiyasu Suzuki. Topographical features of the vagal nerve at the cervical level in an aging population evaluated by ultrasound. *Interdisciplinary Neurosurgery*, 9:64–67, 2017.

[5] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[6] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[7] M. Mrahorovic. Integration of a convolutional neural network for speech-to-text recognition in an fpga compiler flow. Master's thesis, Delft University of Technology, 2021.

[8] Xilinx. https://finn.readthedocs.io/en/latest/internals.html.

[9] https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/.

[10] Andrew Shepard, Bo Wang, Thomas Foo, and Bryan Bednarz. A block matching based approach with multiple simultaneous templates for the real-time 2d ultrasound tracking of liver vessels. *Medical Physics*, 44, 09 2017.

[11] Skanda Bharadwaj, Sumukha Prasad, and Mohamed Almekkawy. An upgraded siamese neural network for motion tracking in ultrasound image sequences. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 68(12):3515–3527, 2021.

[12] Masayuki Shimoda, Youki Sada, Ryosuke Kuramochi, and Hiroki Nakahara. An fpga implementation of real-time object detection with a thermal camera. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 413–414, 2019.

[13] Masayuki Shimoda, Shimpei Sato, and Hiroki Nakahara. Power efficient object detector with an event-driven camera on an fpga. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, HEART 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Hiroki Nakahara and Tsutomu Sasao. A high-speed low-power deep neural network on an fpga based on the nested rns: Applied to an object detector. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[15] Tao Xiao, Yuran Qiao, Junzhong Shen, Qianming Yang, and Mei Wen. Unified virtual memory support for deep cnn accelerator on soc fpga. *Algorithms and Architectures for Parallel Processing Lecture Notes in Computer Science*, page 64–76, 2015.

[16] Heekyung Kim and Ken Choi. The implementation of a power efficient bcnn-based object detection acceleration on a xilinx fpga-soc. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 240–243, 2019.

[17] E. Ben-Menachem. Vagus-nerve stimulation for the treatment of epilepsy.

[18] F.A. Koopman. Vagus nerve stimulation inhibits cytokine production and attenuates disease severity in rheumatoid arthritis. *Proceedings of the National Academy of Sciences*, 113(29):8284–8289.

[19] K.B. Clark. Enhanced recognition memory following vagus nerve stimulation in human subjects. *Nature neuroscience*, 2(1):94–98.

[20] M.P. Neuser. Vagus nerve stimulation boosts the drive to work for rewards. *Nature communications*, 11(1):1–11.

[21] D. Guidolin and A. D. *A New Integrative Theory of Brain-Body-Ecosystem Medicine. Hippocratic Holistic View of Medicine to Our Modern Society*. Int J Environ Res Public Health.

[22] S.e Spuck. Operative and technical complications of vagus nerve stimulator implantation. *Operative Neurosurgery*, 67(suppl$_2$) : 489–494.

[23] E. Ben-Menachem. Vagus nerve stimulation, side effects, and long-term safety. *Journal of clinical neurophysiology*, 18(5):415–418.

[24] J.W. Wheless. Vagus nerve stimulation (vns) therapy update. *Epilepsy Behavior*, 88:2–10.

[25] R.e Kuba. Vagus nerve stimulation: longitudinal follow-up of patients treated for 5 years. *Seizure*, 18(4):269–274.

[26] Katja Hoehn and Elaine Nicpon Marieb. *Human anatomy physiology*. Benjamin Cummings San Francisco, CA, 2010.

[27] Richard Câmara and Christoph J Griessenauer. Anatomy of the vagus nerve. In *Nerves and nerve injuries*, pages 385–397. Elsevier, 2015.

[28] Hippocrates. *On Airs, Waters, and Places*. Hippocrates, Athens, 460 BC–370 BC.

[29] Robert H Howland. Vagus nerve stimulation. *Current behavioral neuroscience reports*, 1(2):64–73, 2014.

[30] T.E. Schlaepfer. Vagus nerve stimulation for depression: efficacy and safety in a european study. *Psychological medicine*, 38(5):651–661.

[31] F.e Marrosu. Vagal nerve stimulation improves cerebellar tremor and dysphagia in multiple sclerosis. *Multiple Sclerosis Journal*, 13(9):1200–1202.

[32] B.V. Bonaz. Vagus nerve stimulation: a new promising therapeutic tool in inflammatory bowel disease. *Journal of internal medicine*, 282(1):46–63.

[33] D.H. Turnbull. Beam steering with pulsed two-dimensional transducer arrays. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 38(4):320–333.

[34] A. Thrush and T. Hartshorne. Vascular ultrasound.

[35] A.D. Chuprov and K. V. *Characteristics of non invasive ultrasound determination of mechanical lens hardness*. Vestnik Oftalmologii.

[36] J. Blackmore and e al. Ultrasound neuromodulation: a review of results, mechanisms and safety. *Ultrasound in medicine biology*, 45(7):1509–1536.

[37] C. Yang and e al. Recent advances in ultrasound-triggered therapy. *Journal of Drug Targeting*, 27(1):33–50.

[38] W.J. Tyler. Noninvasive neuromodulation with ultrasound? a continuum mechanics hypothesis. *The Neuroscientist*, 17(1):25–36.

[39] H.e Kim. Miniature ultrasound ring array transducers for transcranial ultrasound neuromodulation of freely-moving small animals. *Brain stimulation*, 12(2):251–255.

[40] V.e Cotero. Noninvasive sub-organ ultrasound stimulation for targeted neuromodulation. *Nature communications*, 10(1):1–12.

[41] J.e Lee. A mems ultrasound stimulation system for modulation of neural circuits with high spatial resolution in vitro. *Microsystems Nanoengineering*, 5(1):1–11.

[42] W.e Legon. Neuromodulation with single-element transcranial focused ultrasound in human thalamus. *Human brain mapping*, 39(5):1995–2006.

[43] W.e Legon. Transcranial focused ultrasound neuromodulation of the human primary motor cortex. *Scientific reports*, 8(1):1–14.

[44] A.a Priyanka. Towards automated positioning of ultrasonic probes.

[45] Market share android market share. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009. Accessed: 2021-1-2.

[46] G. Dan and S. Nick. Parallel computing for neural networks.

[47] G. Li and Q. W. Imaging-guided dual-target neuromodulation of the mouse brain using array ultrasound. *IEEE Trans Ultrason Ferroelectr Freq Control*.

[48] Christiaan Boerkamp. https://app.roboflow.com/new-workspace-sghkd/vagus-nerve-ultrasound-correctname/1.

[49] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[50] Sabyasachi Sahoo. Deciding optimal filter size for cnns, Nov 2018.

[51] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[52] Christiaan Boerkamp. https://github.com/christiaanboe/wearable-vagus-nerve-object-detector.

[53] Wikipedia contributors. Methods of computing square roots — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-December-2021].

[54] Wikipedia contributors. Long division — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-December-2021].

[55] https://www.slideshare.net/shtaxxx.

[56] Luca Bertinetto, Jack Valmadre, João F Henriques, Andrea Vedaldi, and Philip HS Torr. Fully-convolutional siamese networks for object tracking. *arXiv preprint arXiv:1606.09549*, 2016.

[57] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb 2017.