POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Acceleration of Software Applications on FPGAs using High-Level Synthesis

Candidate

KEVIN DULE

Supervisors

Prof. LUCIANO LAVAGNO

Prof. MIHAI LAZARESCU

PhD Candidate TEODORO URSO

PhD Candidate ROBERTO BOSIO

Institution

POLITO - DET

April 2025

Abstract

We are currently living in a pivotal moment in history, witnessing a technological revolution unfold in real time, as the rise of AI rapidly reshapes the world around us. This disruptive force, known as artificial intelligence, is a field of research in computer science that develops and studies machines and software capable of performing tasks that typically require human intelligence, such as reasoning, problem-solving and language processing. Due to increased data volumes, advanced algorithms, and improvements in computing power and storage, artificial intelligence and machine learning have soared in popularity today, causing extensive research to be dedicated to this field.

Machine learning (ML) is the main field of study of AI, and it focuses on the development and study of statistical algorithms that can learn from data, improve over time, and make predictions or decisions without being explicitly programmed.[1] Neural networks (NN) are a type of machine learning algorithms that teach computers to process data in a way that is inspired by the human brain. These networks use interconnected nodes or neurons in a layered structure that resembles the human brain and how it functions.[2] Activation functions (AF) are types of layers of networks and are used to compute non-linear mathematical expressions. These functions play a critical role in neural networks, as they enable AI/ML models to learn and represent complex patterns in data. However, due to their resource-intensive nature, more efficient solutions are needed.

This brings forth the main purpose of this thesis, which was to develop efficient implementations of the activation functions currently used in the industry and accelerate them on FPGAs. One of the various methods used for this design process is high-level synthesis design, which was also the methodology followed during this thesis, indicatively using Xilinx (AMD) tools, such as Vitis HLS and Vivado. By accelerating these complex non-linear functions and generating networks that are less resource-intensive, devices at the edge (such as IoT devices and embedded systems) would be able to run more complex and advanced AI models for real-time inference.

Acknowledgements

I would like to express my deepest gratitude to all my supervisors, from Professors Luciano Lavagno and Mihai Lazarescu, to PhD Candidates Teodoro Urso and Roberto Bosio, for their invaluable guidance and support throughout my research. I am truly grateful for their mentorship and the trust they placed in me from the very beginning. A special recognition goes to Teodoro for his relentless effort, unwavering support and immense patience in mentoring me. Without the collective contributions of all my supervisors, this work would not have been possible.

Dua të falenderoj çdo person që më ka mbështetur në këtë rrugëtim të gjatë e të vështirë studimesh dhe rritjeje, të përbërë pa dyshim nga sprova e pengesa të shumta që janë prezantuar njëra pas tjetrës, tjetra pas njërës, ndonjëherë edhe njëherësh, me i far pike smerrej mo vesh me thon tdrejten po me nji menyr o tjetren ju dha dum. Dua të falenderoj, duke nisur nga rrethi i ngushtë familjar, mamin, nonën dhe gjyshin, 2 dajat dhe familjet e tyre, duke kaluar më tej tek shoqëria dhe të afërmit, qoftë në Shqipëri, Itali, apo kudo tjetër në botë, e duke përfunduar te çdokush që ka patur sadopak ndikim mbi mua, faleminderit. Ju keni qenë pjesë thelbësore e këtij udhëtimi. Nuk e di se në cilin destinacion do më çojë aventura e rradhës, por e rëndësishmja është se e di që do ta kem gjithmonë mbështetjen tuaj dhe ju timen, e fundja thonë se s'ka shumë rëndësi destinacioni por udhëtimi e bashkudhëtarët.

"Kevin D. Torino 2025"

Table of Contents

Abstract II

Table of Contents VI

List of Tables VIII

List of Figures IX

Acronyms XI

1 Introduction 1
- 1.1 Background 1
  - 1.1.1 Artificial Intelligence 1
  - 1.1.2 Machine Learning 2
  - 1.1.3 Neural Networks 3
  - 1.1.4 Activation Functions 4
  - 1.1.5 Image Classification 4
  - 1.1.6 High-Level Synthesis 5
  - 1.1.7 FPGAs 6
- 1.2 Problem Statement 7
- 1.3 Objectives 8
- 1.4 Contributions 9
- 1.5 Thesis Structure 10

2 Toolchain and Technology 13
- 2.1 Vitis HLS 13
- 2.2 Vivado 14
- 2.3 Visual Studio Code 15
- 2.4 SSH 15
- 2.5 Docker 15
- 2.6 PyTorch 16
- 2.7 ONNX 16

2.8 NVIDIA RTX 4090 GPU . . . 17
2.9 Xilinx® Kria KV260 FPGA . . . 18
2.10 Code::Blocks . . . 19
2.11 Programming Languages . . . 19
2.11.1 C . . . 19
2.11.2 C++ . . . 19
2.11.3 Python . . . 20
2.11.4 HDL . . . 20
2.11.5 HLS . . . 20

3 Methodology 21
3.1 Research Approach . . . 21
3.2 Paper Review . . . 22
3.3 High-Level Synthesis Workflow . . . 22
3.4 Parallelization Strategy . . . 23
3.5 Resource Optimization Techniques . . . 24
3.6 Neural Network Integration . . . 24
3.7 Challenges Encountered . . . 25

4 The Accelerator 27
4.1 Sigmoid . . . 27
4.1.1 Paper Review . . . 28
4.1.2 Implementations . . . 38
4.1.3 Results . . . 47
4.2 SiLU . . . 47
4.2.1 Implementation . . . 48
4.2.2 Results . . . 50

5 Use Case Scenario 51
5.1 ResNet-8 . . . 51
5.1.1 Residual Block . . . 51
5.1.2 Modifying and Training the ResNet-8 Network . . . 52
5.1.3 Integrating the Accelerator . . . 54
5.1.4 Synthesizing and Deploying the model on the Xilinx® Kria KV260 FPGA board . . . 55
5.2 ResNet-20 . . . 56

6 Conclusions 59

Bibliography 61

List of Tables

5.1 Accuracy: Batch Size vs Learning Rate 54
5.2 Resource utilization comparison between ResNet-8 with original ReLU and accelerated SiLU 56
5.3 Performance comparison between ResNet-8 with original ReLU and accelerated SiLU 56
5.4 Accuracy: Batch Size vs Learning Rate 57
5.5 Resource utilization comparison between ResNet-8 with original ReLU and accelerated SiLU 58
5.6 Performance comparison between ResNet-20 with original ReLU and accelerated SiLU 58

List of Figures

1.1 Hierarchy of AI[6] . . . 2
1.2 AI through the years[3] . . . 2
1.3 A biological neuron and an ANN neuron[7] . . . 3
1.4 Structure of a deep neural network[8] . . . 4
1.5 Image classification example[9] . . . 5
1.6 High-level synthesis flow[10] . . . 6
1.7 FPGA block diagram[11] . . . 6
1.8 FPGA board Xilinx® Kria KV260[12] . . . 7

2.1 Vitis HLS stack[13] . . . 13
2.2 Vivado Waveform Viewer . . . 14
2.3 A PyTorch workflow[14] . . . 16
2.4 ONNX interoperability[15] . . . 17
2.5 NVIDIA RTX 4090[16] . . . 17
2.6 Performance comparison between different device types[17] . . . 18

4.1 Standard logistic function[25] . . . 28
4.2 Sigmoid function and its PWL approximation . . . 30
4.3 Sigmoid function and its polynomial approximation . . . 31
4.4 The absolute errors of different PWL function divisions . . . 32
4.5 PWL functions for the 25 intervals . . . 33
4.6 PWL approximation with 3 intervals . . . 34
4.7 Proposed circuit implementation of sigmoid activation function . . . 34
4.8 Block diagram of the proposed sigmoid hardware . . . 35
4.9 First 6 quantized samples of the proposed LUT . . . 35
4.10 Workflow of modular approximation methodology NRA . . . 36
4.11 Logarithmic approximation of the sigmoid . . . 37
4.12 Comparison of different hardware designs . . . 38
4.13 Synthesis report of the original sigmoid function . . . 39
4.14 Synthesis report of sigmoid v1 with no optimizations . . . 40
4.15 Synthesis report of sigmoid v1 after optimizations . . . 41
4.16 Synthesis report of sigmoid v2 with no optimizations . . . 41

4.17 Synthesis report of sigmoid v2 after optimizations 42
4.18 Synthesis report of sigmoid v3 with no optimizations 42
4.19 Synthesis report of sigmoid v3 after optimizations 43
4.20 Synthesis report of sigmoid v4 after first optimizations 44
4.21 Sigmoid function approximation using a LUT with 128 entries of 8 bits with range $[0, 8]$ 44
4.23 Zoomed in version of Figure 4.21 44
4.24 Zoomed in version of Figure 4.22 44
4.22 Sigmoid function approximation using a LUT with 256 entries of 8 bits with range $[0, 8]$ 45
4.25 Synthesis report of sigmoid v5 implemented in LUTRAM 46
4.26 Synthesis report of sigmoid v5 implemented in BRAM 46
4.27 SiLU and ReLU activation functions[38] 48
4.28 SiLU function approximation using a LUT with 256 entries of 8 bits for input range of [-16, 15.875] 49
4.29 Zoomed in version of Figure 4.28 49

5.1 A ResNet residual block displaying a residual connection[39] 52

Acronyms

PhD - Doctor of Philosophy

FPGA - Field Programmable Gate Array

POLITO - Politecnico di Torino

DET - Department of Electronics and Telecommunications

AI - Artificial Intelligence

ML - Machine Learning

DL - Deep Learning

NN - Neural Network

AF - Activation Function

HLS - High-Level Synthesis

AMD - Advanced Micro Devices

LUT - Look-Up Table

DNN - Deep Neural Network

LLM - Large Language Model

CNN - Convolutional Neural Network

GAN - Generative Adversarial Network

ReLU - Rectified Linear Unit

SiLU - Sigmoid Linear Unit

ANN - Artificial Neural Network

ViT - Vision Transformer

ResNet - Residual Network

VHDL - VHSIC HDL

HDL - Hardware Description Language

VHSIC - Very High-Speed Integrated Circuit

RTL - Register-Transfer Level

ASIC - Application-Specific Integrated Circuit

CLB - Configurable Logic Block

DSP - Digital Signal Processing

CPU - Central Processing Unit

GPU - Graphics Processing Unit

LSB - Least Significant Bit

FF - Flip-Flop

VS Code - Visual Studio Code

SSH - Secure Shell

Tmux - Terminal Multiplexer

ONNX - Open Neural Network Exchange

VPU - Vector Processing Unit

RTX - Ray Tracing Texel eXtreme

IDE - Integrated Development Environment

Cosim - Co-Simulation

SoC - System on Chip

CC - Clock Cycle

FIFO - First In First Out

BRAM - Block RAM

RAM - Random Access Memory

LUTRAM - Look-Up Table Random Access Memory

PWL - Piecewise Linear

VLSI - Very Large Scale Integration

LSTM - Long Short-Term Memory

NR - Newton-Raphson

FC CORDIC - Fast-Convergence Coordinate Rotation Digital Computer

II - Initiation Interval

Tanh - Hyperbolic Tangent

GELU - Gaussian Error Linear Unit

YOLO - You Only Look Once

LR - Learning Rate

SGD - Stochastic Gradient Descent

CP - Critical Path

ILP - Integer Linear Programming

FPS - Frames Per Second

1. Introduction

This thesis conducts a thorough research on the acceleration of software applications on FPGAs using High-Level Synthesis. The target applications are activation functions, which are instrumental components in the fields of artificial intelligence and machine learning. Accelerating these functions means generating resource-efficient hardware solutions on FPGAs, freeing up resources that can be allocated to other computationally intensive components, thus allowing for more advanced AI models to be run on the same boards.

The final product of this work is a generalized LUT-based method of implementing activation functions, which can be used to accelerate many different univariate functions, and can be easily integrated into neural networks. The findings of this thesis offer valuable contributions to academia and industry, facilitating further advancements in FPGA-based neural network acceleration.

This first chapter serves as an introduction to the following body of work, by glancing through a few brief sections that give insight into relative topics such as the general background of concepts discussed later on, statement of the problem aimed to be tackled, main objectives of the thesis, resulting contributions to academia and industry and finally the layout of the subsequent chapters of this document.

1.1 Background

1.1.1 Artificial Intelligence

The term artificial intelligence, which was originally coined in 1956,[3] refers to all the computational systems that are capable of handling tasks that usually require human intelligence, such as learning, perception, and decision-making.[4] AI research started in the 1950s, with its birth attributed to the groundbreaking paper Alan Turing published in 1950, titled "Computing Machinery and Intelligence", where he proposed the famous Turing Test to evaluate whether a machine can exhibit intelligent behavior indistinguishable from that of a human.[5]

Extensive research was conducted through the second half of the 20th century, as can be seen in Figure 1.2, with continuous breakthroughs in the development of algorithms and computing devices. Today, LLM-powered AI assistants are being integrated into many existing apps we commonly use in everyday life.

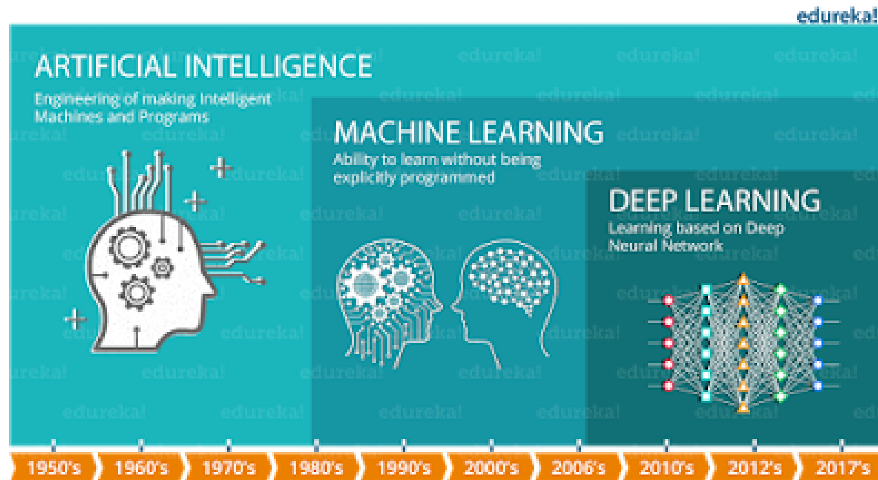

Figure 1.1: Hierarchy of AI[6]

1.1.2 Machine Learning

Machine learning has always been considered a central pillar of AI since the early decades, as seen in Figure 1.2, and it fundamentally consists of algorithms and mathematical models whose duty is to perform the learning part of AI using a given dataset, and then be able to generalize enough in order to handle even unseen data.

Figure 1.2: AI through the years[3]

ML is such a versatile method that its use can vary from speech recognition to computer vision, with applications in different fields, from agriculture to

medicine. Various models have been created through the years, each with their own strengths and weaknesses, so choosing the most optimal model for a specific task is fundamental. Examples of such models include regression analysis, artificial neural networks, decision trees, and support-vector machines.

1.1.3 Neural Networks

As mentioned above, a neural network is one of the mathematical models used in machine learning. Its architecture is inspired by the human brain, which is composed of brain cells, also called neurons. These form an intricate, interconnected network where neurons are able to communicate with each other by sending signals in the form of electric impulses (action potentials). Neural networks try to mimic this structure, as illustrated in Figure 1.3, using software to generate nodes, which receive data as an input, compute some mathematical expression, and produce an output to be sent to the next node.

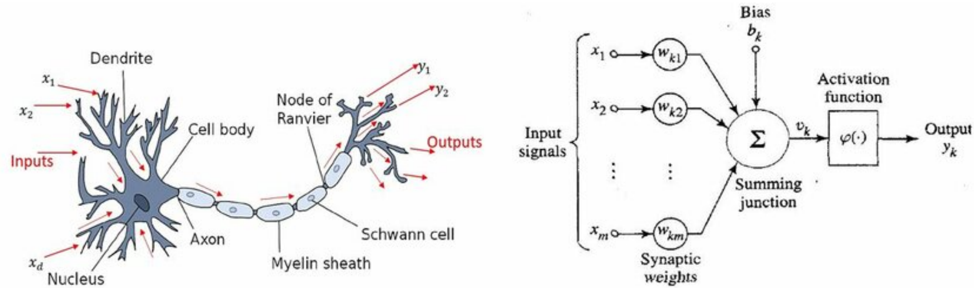

Figure 1.3: A biological neuron and an ANN neuron[7]

Each neuron computes a weighted sum, as seen in (1.1), before passing the value to an activation function.

$$y_i = \sum_{j=1}^{n} w_{ij} \cdot x_j + b_i \tag{1.1}$$

Any layer that is between the input and output layers of a network, and does not have direct contact with external data, is called a hidden layer. A network consisting of several hidden layers with millions of neurons is considered a deep neural network, as depicted in Figure 1.4. Some commonly used networks include convolutional neural networks (CNNs), generative adversarial networks (GANs), and transformer networks. DNNs typically need to undergo much more training in exchange for better performance, especially at complex tasks, as they can theoretically map any input type to any output type.

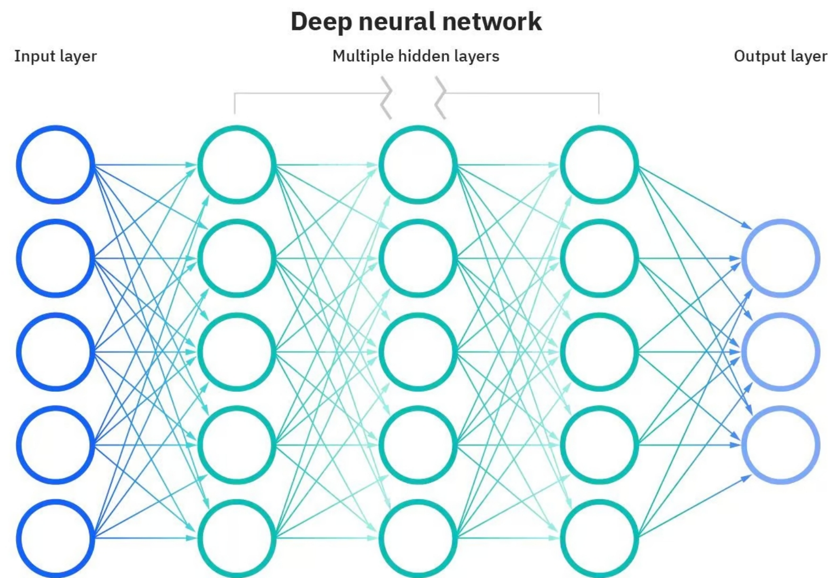

Figure 1.4: Structure of a deep neural network[8]

1.1.4 Activation Functions

Activation functions are non-linear mathematical expressions and constitute an integral part of neural networks. They introduce non-linearity to the model, improving their performance greatly, by enabling them to model more complex relationships between data. Without these functions, neural networks would model solely linear relationships, through linear operations such as matrix multiplication. Thus, they are typically placed between this summation junction and the output of the neuron, as can be seen in Figure 1.3.

Given that most real-world data and their relationship is not linear, networks would fail to learn these complex input-output mappings. Designers can choose from a wide pool of activation functions, such as the rectified linear unit (ReLU), the standard logistic function (Sigmoid), or the sigmoid linear unit (SiLU).

1.1.5 Image Classification

Image classification is one of the tasks that can be handled with the use of AI, and more specifically, deep neural networks. It consists of the network receiving an image as an input, computing calculations, and finally generating as the output a prediction of what is depicted in the image, as shown in Figure 1.5.

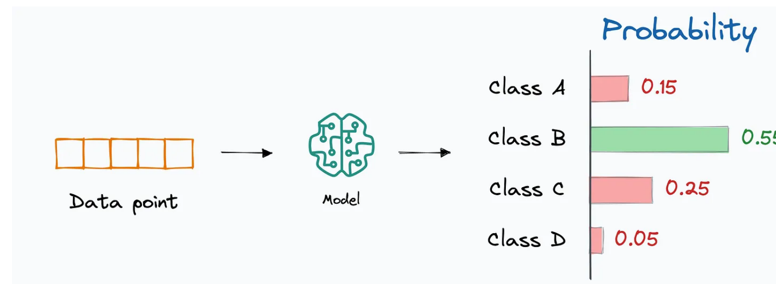

Figure 1.5: Image classification example[9]

When these models are trained, they are given a large dataset containing a vast amount of images and a list of classes. Each of the images will have as a label one of the classes, describing the content of the image. After being trained, the model can receive as an input an image and it will produce the relative probabilities that the image is any of the given classes. Deep neural networks are a powerful solution when dealing with image classification tasks, with ResNet, MobileNet, and Vision Transformers (ViTs) being some of the viable choices.

1.1.6 High-Level Synthesis

High-Level Synthesis is a design methodology in hardware development where the code is written using high-level programming languages (C, C++ or SystemC), instead of hardware description languages (VHDL or Verilog). HLS tools are able to compile this code and generate an RTL, as well as the HDL code. This makes the code written by the designer much more readable and less complex, reducing development time and allowing for better design space exploration during the same time frame. Designers are still able to manually affect the hardware design at the lowest level, by applying specific constraints in the code, which will be taken into consideration by the compiler during synthesis. In this way, arrays, loops, or functions can be manipulated using directives such as memory partitioning, pipelining, or loop unrolling.

High-level synthesis can be incredibly powerful when targeting FPGAs or ASICs due to their inherently customizable structures, allowing for seamless hardware/software co-design. For these reasons, big tech companies have developed their own proprietary tools to enable designers to create HLS solutions, with software such as Vitis HLS (AMD, Xilinx), Intel HLS Compiler (Intel), Cadence Stratus HLS (Cadence) or Mentor Catapult HLS (Siemens EDA).

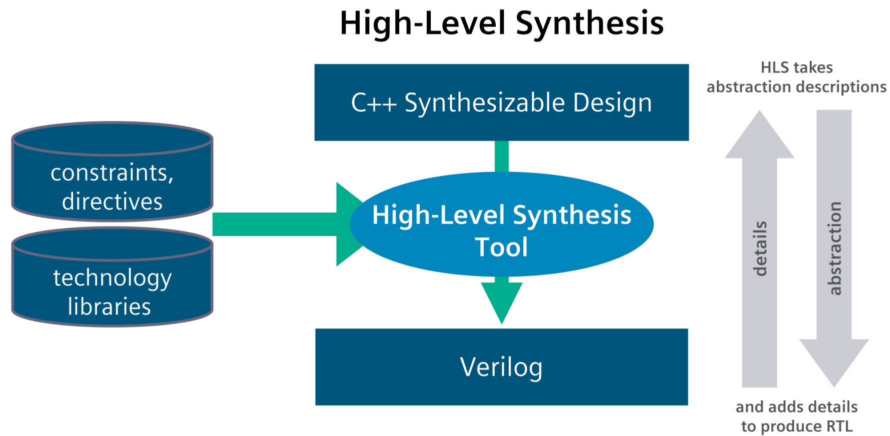

Figure 1.6: High-level synthesis flow[10]

1.1.7 FPGAs

FPGAs (Field-Programmable Gate Arrays) are flexible, reconfigurable chips that let engineers design custom digital circuits. Unlike traditional processors that run software sequentially, FPGAs use a grid of programmable logic blocks, called CLBs, and shown in Figure 1.7, to perform multiple tasks in parallel. This makes them ideal for high-speed applications like AI acceleration, real-time data processing, and signal processing.

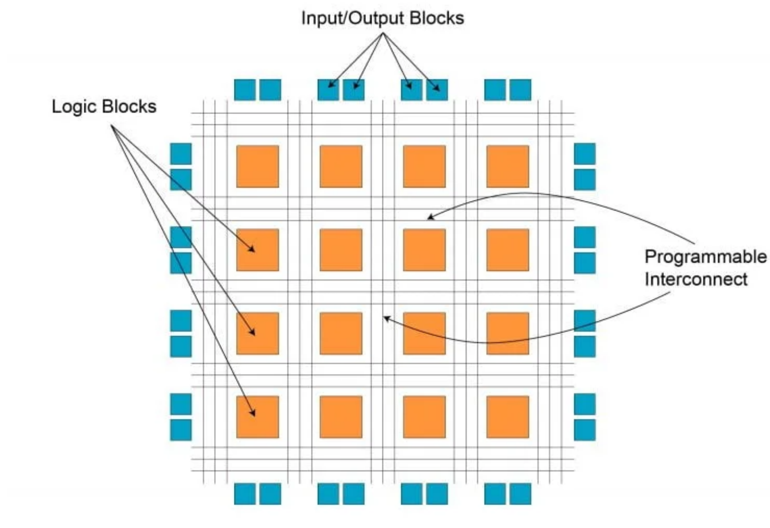

Figure 1.7: FPGA block diagram[11]

One major advantage of FPGAs is their ability to be reprogrammed after manufacturing, making them useful when deployed in rapidly evolving industries or when utilized for prototyping and researching. They often outperform CPUs in specialized tasks, delivering higher efficiency and lower latency. However, programming FPGAs requires expertise in hardware description languages (HDLs) like Verilog or VHDL, making development more complex than traditional software.

While FPGAs can be more expensive and power-hungry than ASICs in large-scale production, advancements in tools like Vitis HLS are making them more accessible. These tools allow engineers to design using high-level languages like C++, making hardware/software co-design more seamless. Despite the challenges, FPGAs remain a powerful choice for applications where speed, flexibility, and efficiency are key.

Figure 1.8: FPGA board Xilinx® Kria KV260[12]

1.2 Problem Statement

Activation functions are non-linear mathematical expressions, meaning they can be very resource-intensive when implemented in hardware. Complex operators such as divisions, exponentials, or logarithms might be needed to be calculated, requiring costly hardware units like DSPs to be allocated. These resources are typically a significant constraint when designing hardware solutions, due to their

scarcity in semiconductor devices. This makes the importance of accelerating them quite evident.

The high cost of activation functions in terms of hardware, limits the advancement of neural networks. A model could achieve on paper a better performance using a better activation function, but it wouldn't be possible to deploy it on a physical board. Implementing an accelerated version of those activation functions could allow the model to be deployed on the board, with a possible trade-off in accuracy.

Moreover, managing to transform activation functions from DSP-hungry expressions into less demanding ones, can free up a lot of resources that can be allocated to other heavy-load layers on the same network, such as convolutional layers, increasing overall performance of the model.

Another critical issue to be taken into consideration, is the limitations of using CPUs or GPUs to implement neural networks. CPUs' strongest suit is operating in a sequential fashion, which is not optimal for neural networks, since most of the workload for these networks consists of computing matrix multiplications. These operations are highly parallelizable, and FPGAs, due to their structure, can handle that more efficiently. GPUs can solve the issue of parallel execution as well, however they are power-hungry, so FPGAs can usually achieve higher performance per watt. Their configurability enables maximal possible optimization, reducing power consumption significantly. Lastly, FPGAs can provide ultra-low latency and high throughput, outperforming CPUs and GPUs in real-time applications, the domain where AI models are typically needed.

1.3 Objectives

The primary purpose of this thesis is to generate optimized implementations of activation functions on FGPAs. In specific, the objectives of this thesis are as follows:

- Develop a generalized method to accelerate various univariate activation functions on FPGAs using HLS
- Design scalable parallel architectures to ease the integration of the created IPs into larger systems
- Implement some activation functions using this method and synthesize them

- Integrate and test these accelerators on pre-existing neural networks of different sizes and parallelisms
- Deploy on a physical board the bitstream of the modified networks with the new accelerators and compare the real inference results to the simulated ones
- Optimize the trade-off between performance and resources

1.4 Contributions

This thesis addresses the gap in research regarding the implementation of activation functions using look-up tables (LUTs), by doing an in-depth study on this alternative approximation method and offering a generalized solution for it. The key contributions of this research can be subdivided into three main branches: one concerning the hardware blocks designed on Vitis HLS and representing the accelerated activation functions handling one input at a time, one concerning the overall impact on the neural networks using numerous instances of it, and a final one with general remarks. They are detailed as follows:

1. Accelerator on Vitis HLS:
   - Up to 97% decrease in LUT and FF utilization per activation function
   - No DSPs allocated
   - Decrease in latency by a factor of 10
   - No errors introduced when working with 8-bit values
   - When compared to 32-bit floating point values, the maximum possible error introduced is $\frac{1}{2}$LSB of the 8-bit values stored in the LUTs and much smaller average errors
2. Neural network deployed on FPGA board:
   - Improved accuracy of ResNet-8 by 1.02%
   - Improved accuracy of ResNet-20 by 2.69%
   - Virtually the same throughput and power consumption
   - Minor trade-offs in hardware resource utilization
3. General remarks:
   - The method is general, meaning other engineers in the field can apply it to implement other univariate activation functions not mentioned here
   - Offering new insights in a field with not enough detailed research

- Implemented an HLS-based workflow targeting FPGAs, thus reducing development time and creating a flexible and reconfigurable final product
- Resulting hardware solution is adaptable and scalable

These contributions provide a foundation for advancements in FPGA-based neural network implementations, with potential adoption in real-time AI applications, such as edge computing and embedded systems. They also serve as a basis for future research in the same field.

1.5 Thesis Structure

This thesis is organized into chapters, sections, and subsections. Each chapter is divided into sections, and sections are further subdivided into subsections. For instance, the numbering '1.2.3' refers to Chapter 1, Section 2, and Subsection 3. The chapter outline of this thesis is as follows:

- Chapter 1: Introduction

  This chapter introduces the body of work, provides the background, and states the problem addressed in the thesis. It also outlines the objectives, contributions, and the structure of the document.

- Chapter 2: Toolchain and Technology

  This chapter discusses the various tools and technologies used in the research, including the development environment and the hardware setup, as well as the relevant programming languages and frameworks such as C++, Python, and PyTorch.

- Chapter 3: Methodology

  In this chapter, the research approach and methodology are explained. It covers the high-level synthesis workflow, parallelization strategy, resource optimization techniques, and the integration into real neural networks. The chapter also discusses the challenges faced during the research process.

- Chapter 4: The Accelerator

  This chapter provides a detailed description of the proposed accelerator, including the implementation of activation functions (Sigmoid and SiLU). It also reviews related papers, implementation strategies, and presents the results of the designed accelerator.

- Chapter 5: Use Case Scenario

  This chapter presents real-world use cases of the accelerator, including experiments with ResNet8 and ResNet20, demonstrating the practical application and integration of the accelerator in deep learning tasks.

- Chapter 6: Conclusions

  This chapter summarizes the research work, discussing the practical applications of the final product, highlighting the key findings and achievements, the limitations of the proposed approach and the possible future directions in which this research could extend.

- Appendices and Bibliography

  The appendices provide supplementary material, including additional data, figures, and code. The bibliography lists all the references cited throughout this document.

2. Toolchain and Technology

This chapter aims to equip the reader with a clear understanding of the various tools, technologies, and software used throughout the development of this thesis. A brief description for each of the software tools and hardware devices used during this process will be given, highlighting the key benefits they offered to the overall development environment.

2.1 Vitis HLS

Vitis High-Level Synthesis (HLS) is a tool that allows developers to design hardware accelerators using high-level programming languages like C and C++. Instead of writing low-level hardware descriptions, engineers can design complex hardware blocks while also producing a more readable code, making hardware development more accessible. This tool automates the conversion of high-level code into hardware descriptions, as illustrated in Figure 2.1, while offering various optimizations to improve speed and resource efficiency.

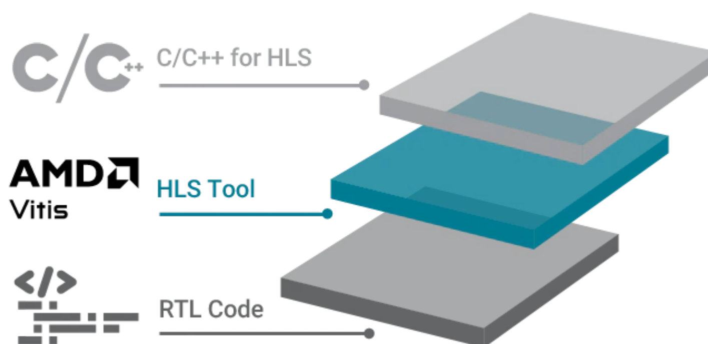

Figure 2.1: Vitis HLS stack[13]

This is the tool used in this thesis to design the optimized implementations

of the activation functions, by mainly benefiting of the hardware and software co-design capabilities, such as writing hardware-aware C++ code or applying directives and pragmas to the code. Vitis HLS is particularly useful for FPGA-based acceleration, as it simplifies the process of creating and testing custom hardware components before integrating them into a larger system.

2.2 Vivado

Vivado is an FPGA design suite that provides tools for synthesizing, implementing, and verifying digital circuits. It takes hardware descriptions and maps them onto FPGA architectures, ensuring efficient use of resources. The software includes optimization features that help improve performance while meeting timing and power constraints. It also offers simulation and debugging capabilities, making it easier to verify that a design functions correctly before deployment. Vivado works seamlessly with Vitis HLS, allowing developers to take high-level designs and turn them into optimized FPGA implementations.

After designing and testing something on Vitis HLS, the following step typically was to synthesize, implement and verify on Vivado, in order to compare resources, performance and overall functionality. Viewing and analyzing signal waveforms on Vivado, like the ones in Figure 2.2, was an incredibly useful tool, especially for debugging. Lastly, Vivado is responsible for creating the bitstream, which is the final element of the design, and precisely the one uploaded to the physical FPGA board, in order to effectively deploy the model for real-world inference.

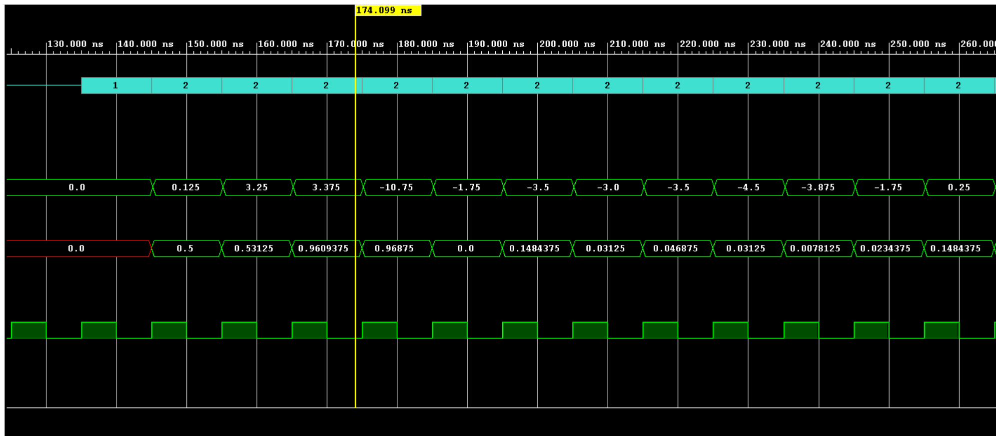

Figure 2.2: Vivado Waveform Viewer

2.3 Visual Studio Code

Visual Studio Code (VS Code) was the principal development environment for the experiments conducted on the neural networks. This editor served as the main tool for all the training, modifications, and testing done on the networks. Its lightweight design, combined with powerful extensions, made it ideal for writing and managing code efficiently. Its built-in terminal was used to connect to remote machines, edit scripts, and monitor logs in real time. The ability to use SSH in it was quite advantageous, providing a simple way to connect to remote servers.

2.4 SSH

Secure Shell (SSH) was essential for remote access to the machines running the workloads. SSH was used to connect to multiple servers and virtual machines equipped with GPUs and FPGAs, which were instrumental in carrying out the training and deployment of AI models entirely remotely, eliminating the need to be physically present at the hardware. To ensure uninterrupted work, tmux was also utilized, allowing the sessions to keep running even if the internet connection was lost. This made it easy to reconnect and resume experiments without losing progress. Through SSH and tmux, it was possible to transfer files, launch training scripts, and monitor resource usage efficiently, enabling a smooth and reliable workflow, with the ability for remote troubleshooting.

2.5 Docker

Docker played a crucial role in maintaining a consistent and portable development environment. Once connected to the remote machines, Docker containers that encapsulated all the necessary dependencies for working with neural networks, were ran. This eliminated compatibility issues and ensured that the software environment remained stable across different systems. Whether it was training, testing, or deploying models, Docker allowed switching between different setups effortlessly, making experimentation and development more streamlined. Opening multiple terminals, and running docker containers in them, enabled multiple trainings or syntheses of networks to be run in parallel, greatly improving efficiency and reducing the total time needed.

2.6 PyTorch

PyTorch was the primary deep learning framework used in this thesis for developing, training, and refining neural networks, offering a flexible and efficient environment for deep learning. It eased the process of implementing architectures like ResNet-8 and ResNet-20, and experimenting with different hyperparameters, including learning rates, batch sizes, optimizers, and schedulers.

This framework's versatility allowed for modifications of the structures of the neural networks, for instance the replacement of activation functions with other ones or the addition of quantization and de-quantization blocks to aid quantization-aware training. With its extensive library support, PyTorch enabled rapid prototyping and fine-tuning of models. Once trained, the models were evaluated for accuracy and performance before being converted to ONNX format for further testing and deployment. A typical workflow using PyTorch can be seen below in Figure 2.3

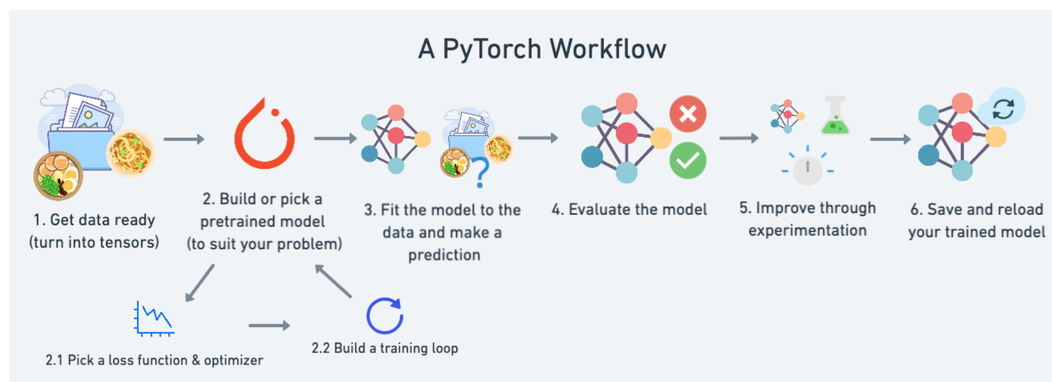

Figure 2.3: A PyTorch workflow[14]

2.7 ONNX

The trained AI models were exported to the ONNX (Open Neural Network Exchange) format to ensure compatibility with different deployment environments. ONNX provided a standardized way to represent deep learning models, making it easier to integrate them into various frameworks and tools. This conversion was crucial for testing the models on non-PyTorch platforms and generating optimized C++ code for FPGA deployment. ONNX Runtime proved to be a key tool, not only for analyzing the model's inference performance, but also for debugging them. Additionally, the ONNX model served as an intermediate representation to bridge the gap between software training and hardware implementation, facilitating an

efficient deployment on the FPGA board, by leveraging the generated C++ HLS code and Vivado tools to produce the bitstream to be uploaded on the board.

Figure 2.4: ONNX interoperability[15]

2.8 NVIDIA RTX 4090 GPU

The NVIDIA RTX 4090 GPU played an important role in this research, being connected to remote machines and utilized for both training the deep learning models and testing their performance by running ONNX runtime inference. With its outstanding computational power, the RTX 4090 significantly sped up the training process, allowing for faster experimentation and fine-tuning of hyperparameters.

Figure 2.5: NVIDIA RTX 4090[16]

High-end GPUs excel at handling the parallel computations required by DNNs, making them ideal for complex models like ResNet-20, while also leveraging larger datasets. This parallelism allows for multiple trainings to be run concurrently, meaning faster design space exploration. Moreover, its numerous cores and high memory bandwidth ensured quick and efficient inference. Figure 2.6 displays the performances of different types of devices and it can be seen that GPUs (e.g. Nvidia V100) can typically achieve the highest throughputs, making them the best choice for pre-deployment phases, and in particular training, prototyping, and testing.

	GFLOP/s	W	GFLOP/J
Xeon Phi	2000	300.0	10
Movidius X	12	**0.3**	40
Nvidia V100	**15000**	200.0	75
Nvidia Xavier	1400	30.0	46
Virtex US+	**10000**	20.0	**500**
Zynq US+	800	10.0	80
Apple A14	1000	6.0	**166**
Kirin 970	2000	6.0	**333**

Figure 2.6: Performance comparison between different device types[17]

2.9 Xilinx® Kria KV260 FPGA

The Xilinx® Kria KV260 FPGA board, which is illustrated in Figure 1.8 served as the target platform for the deployment of the definitive deep neural network model, enabling real-time inference with optimized hardware acceleration. FPGAs offer customizable architectures that can be tailored to specific workloads, allowing for efficient execution of neural networks while minimizing power consumption.

FPGAs are a very powerful option for edge computing and embedded applications, where low latency and energy efficiency are critical. By implementing the trained model on the KV260, it was possible to evaluate the real-world performance of the modified DNNs, with the accelerated activation functions, on a real physical device. Figure 2.6 highlights the trade-offs between different computing platforms, showing that while GPUs excel in training, FPGAs provide the highest performance

per watt, whilst still achieving the second best throughput, making them an ideal choice for deployment.

2.10 Code::Blocks

Code::Blocks is a lightweight and versatile integrated development environment (IDE). The primary use for this IDE was initial code development, algorithmic exploration, and functional testing. This step allowed for rapid prototyping and functional correctness checking, using C/C++ before translating the algorithms into high-level synthesis constructs containing hardware-specific transformations and optimizations. This approach provided a structured and iterative workflow, enabling quick debugging and validation of logic before moving to the more resource-constrained and hardware-aware development environment of Vitis HLS.

2.11 Programming Languages

The development process in this research involved the use of multiple programming languages, each serving a specific role in the algorithm design, training, and hardware implementation processes. These languages provided the flexibility and performance required for various stages of the project, from high-level algorithm development to low-level hardware synthesis.

2.11.1 C

C is a powerful, high-performance programming language which, in this thesis, was primarily used for algorithm development and prototyping on Code::Blocks. Its efficiency and low-level control made it an ideal choice for writing the core logic and computational steps of the activation functions. C provided the flexibility to implement custom algorithms and fine-tune them before they were moved to a higher-level or hardware-specific environment. Its simplicity allowed for quick development and debugging during the early stages of the project, ensuring a solid foundation for the subsequent FPGA-based implementation.

2.11.2 C++

C++ was used in the Vitis HLS environment for more complex algorithm development and for integrating advanced features such as object-oriented programming and libraries that enhanced code performance. In addition to algorithmic design, C++ was also used for preparing the code to be synthesized in Vitis HLS and Vivado. This language's rich feature set, including templates, classes, and

special datatypes, enabled the development of highly optimized, modular, and scalable solutions for complex algorithms. C++ was especially valuable when dealing with HLS specific features such as parallelism, unrolling, or pipelining, by allowing the integration of directives and pragmas into the code.

2.11.3 Python

Python, with PyTorch as a framework, played a critical role in deep neural network development and training. Python's simplicity and flexibility allowed for rapid experimentation with various model architectures. PyTorch provided a powerful platform for training, optimizing and exporting neural networks, making Python the primary language for working on the models. Python was also integral to evaluating and testing the trained models through inference with ONNX runtime. This environment facilitated the manipulation of datasets, model architectures, and training parameters, which were critical for the fine-tuning process.

2.11.4 HDL

Although HDL code was not directly written during the research, hardware descriptions for the FPGA implementations were still generated by Vitis HLS. These descriptions were useful to understand how the software synthesized in some cases specific lines of code. Vitis HLS converted the high-level C++ code (HLS) into HDL structures, as if they were designed using VHDL or Verilog, and were then synthesized for the Xilinx® Kria KV260 FPGA board using Vivado. This process streamlined the design of the custom hardware accelerators, efficiently mapping them to FPGA hardware.

2.11.5 HLS

Vitis HLS was used to convert the C++ code into hardware-specific RTL code. High-level synthesis bridges the gap between high-level software programming and low-level hardware design, enabling FPGA acceleration for software applications, such as activation functions. Vitis HLS allowed for the optimization of the C++ codes for performance, resource utilization, and energy efficiency on the FPGA. This was done by applying certain directives or adding pragmas into the codes, with specific optimization purposes. Some examples involve pipelining functions or instructions, unrolling loops, and binding arrays to specific types of memories. By generating efficient hardware designs from high-level code, it was possible to create custom accelerators that significantly outperformed traditional CPU and GPU implementations.

3. Methodology

This section describes the methodology followed during the completion of this body of work, from initial concepts and sketches to final implementations and optimizations. This methodology can be split into two major phases, a research one and an implementation one. The first phase (research phase) consisted mainly of reading material on all the topics related to this thesis, in order to gain more familiarity and confidence with the terminology and concepts, subsequently allowing for a more technical literature review to be done. After finishing the research phase, the second phase (implementation phase) was up next, and the methodology discussed for this phase regards mainly the overall design flow and techniques used, such as the optimization and parallelization strategies, or the integration into real neural networks.

3.1 Research Approach

The main field of research for this thesis is artificial intelligence, so the first preparatory materials read, were on AI and all its sub-fields, such as machine learning, neural networks and activation functions, in a hierarchic manner.

The very first papers read were the following: the efficient processing of deep neural networks[18], a comprehensive survey of 400 activation functions for neural networks[19], FPGA implementation of sigmoid and hyperbolic tangent activation functions in an artificial neural network[20], and efficient implementation of the softmax activation function on FPGAs[21]. These papers provided a strong background in the main concepts, and the essential foundation needed to make an educated decision on what publications to read next.

Moreover, various materials on the high-level synthesis field were studied, discussing concepts such as parallel programming[22], or the vast amount of topics detailed in the Vitis HLS user guide[23]. Additionally, documentations of hardware specifications for devices such as SoC boards[24], was read. After some practice on the newly encountered concepts of parallel programming and HLS design, the

following step was to start doing paper review on the first activation function to be accelerated, studying all various possible approaches to this problem.

3.2 Paper Review

A thorough literature review was conducted to explore existing acceleration methods for activation functions on FPGAs, with the first selected function being the logistic function, otherwise known as the sigmoid function due to its characteristic S-shaped curve. After this review, the most promising methods were selected and summarized for comparison. These approaches were then implemented, optimized, and evaluated to identify the best candidate for acceleration, as Chapter 4 will explain in more detail.

3.3 High-Level Synthesis Workflow

This section will outline the overall HLS design workflow, starting from an idea or concept up to the final bitstream generated. This workflow can be divided into two major workflows, a code design one and a Vitis HLS environment one.

The first workflow starts with an idea or a concept, which can be a product of paper review or brainstorming. A first structural sketch on paper is drawn, outlining the main elements and blocks of the design, how they are connected, and what the dataflow is. This allows for some pseudo-code to be written, translating the block diagram on paper into general instructions. After creating a clear logical structure of the design and its functionality, the pseudo-code can be developed into C code in an IDE (e.g. Code::Blocks as explained in the previous chapter). The algorithm is fully implemented, tested for functional correctness and analyzed for possible optimizations. Finally, the code can be transformed into C++ code, written in Vitis HLS, allowing for more hardware-aware optimizations to be done, applying directives and adding pragmas to the code. At this point the algorithm is ready to undergo the next workflow.

The second workflow is dictated by the Vitis HLS and Vivado work environments. The fully implemented and optimized hardware design will undergo the processes that are encompassed by the HLS environment, namely Simulation, Synthesis, C/RTL Cosim, Packaging, Implementation, and Bitstream Generation.

- The first process simulates the high-level design to verify functionality and ensure the software behaves as expected before hardware implementation.

- The second one translates high-level code into a hardware description, optimizing it for the target FPGA architecture.

- The third one integrates both hardware and software models to validate the interaction between the hardware design and the software running on the FPGA.

- The fourth one involves preparing the design files and necessary resources for deployment onto the target FPGA, creating a deployable package.

- The fifth one optimizes and places the synthesized design on the FPGA, mapping it to specific resources while ensuring performance and timing constraints are met.

- The sixth, and final one, compiles the design into a binary file that configures the FPGA hardware, leading to deployment.

3.4 Parallelization Strategy

Deep neural networks involve numerous operations that can be computed in parallel, so creating a design exploiting this fact, is instrumental. This section will give a brief overview of the main elements that comprise the parallelization strategy followed in this body of work. At the lowest level, loop unrolling is applied, enabling the concurrent execution of multiple iterations of a loop. This typically increases the code size and resources utilized in order to handle more instructions in parallel. Pipelining is then applied, to overlap different stages of instruction execution, allowing the system to work concurrently on multiple instructions that are at different execution stages.

Given that the block at a lower level can now handle numerous new instructions at each clock cycle, it also need to be fed enough data at each CC to avoid deadlocks or becoming a bottleneck. Thus, at a higher level, a number of HLS streams (FIFOs) in parallel, are used to pass structures containing arrays of vectors of data, to match the inner parallelization of the block, increasing the overall data throughput. All these techniques are parametrized in the code, making this design scalable, and able to handle diverse levels of parallelization need. This implementation will only synthesize the number of blocks needed by the network at different points, with the requested amount of parallelization to achieve preset performance requirements.

3.5 Resource Optimization Techniques

Efficient utilization of the resources of the FPGAs is crucial when deploying large AI models. These models contain resource-intensive blocks like convolution layers, so using more complex, but high performance, activation functions becomes difficult. Accelerating these activation functions makes them viable choices, when discussing real-world deployment. To do this, many resources optimization techniques were utilized, manually handling resource usage at the lowest level (LUTs, FFs, BRAMs, and DSPs).

One key optimization in this design is the way the arrays are implemented. Their size is always a power of two, which aligns well with FPGA memory structures and simplifies address calculations. These arrays are specifically mapped to LUTRAM instead of BRAM, reducing the use of block memory resources and allowing them to be allocated to other critical operations, such as storing the large amounts of weights and biases needed for the convolution layers. The design operates on 8-bit values, ensuring that computations remain aligned with standard byte-based hardware structures, leading to better resource utilization and simpler data handling.

To further optimize resource usage, this method avoids completely the use of DSPs which are normally required to compute complex mathematical expressions, such as exponentials, divisions, or logarithms. Instead LUTs are used, increasing tremendously the throughput, due to the way they are implemented in the fabric of the FPGA, whilst also freeing up DSPs, which can be allocated to convolution layers. Additionally, while the code includes many different tables, with the way it is designed, only one instance is synthesized based on actual requirements, minimizing the overall hardware footprint. These optimizations collectively contribute to an efficient, scalable, and resource-conscious hardware implementation.

3.6 Neural Network Integration

To evaluate the impact of the accelerated block on both performance and accuracy, it was integrated into real neural networks. The existing network architecture was modified by replacing the original activation function with a more advanced one, it was trained to the best accuracy, and then the advanced activation function was replaced by the newly developed hardware-accelerated implementation. The first replacement improved performance, but the resource demand sky-rocketed, making it impossible to be deployed on a physical board. The second replacement decreased substantially the resource demand, making it a feasible solution now,

with no losses in accuracy, and still guaranteeing overall improvement.

The final, accelerated model was then deployed on the FPGA board, and tested using the same dataset as the software implementation to allow for a direct comparison. Real-world benchmark performances of the board were analyzed to assess eventual trade-offs with the original version when deployed. This approach provided insights into how the accelerated block influences both computational efficiency and overall model reliability, ensuring that the hardware implementation remains both practical and effective for real-world deployment.

3.7 Challenges Encountered

The challenges encountered during this research were numerous, as expected from dealing with such advanced and state-of-the-art concepts. However, through resilience, persistence, and curiosity, they were all successfully overcome. In this section, some of the main issues, and how they were resolved, are discussed.

- Resource Utilization – Efficiently managing FPGA resources such as LUTs, BRAM, and DSPs is crucial to avoid bottlenecks. Solution: Optimizing memory usage by leveraging LUTRAM instead of BRAM, minimizing DSP usage, and applying parallelization techniques selectively to balance performance and resource constraints.

- Quantization vs Accuracy – The accelerated activation function required reduced precision, which affected accuracy. Solution: Choosing an optimal fixed-point or quantized representation (e.g., 8-bit) to maintain acceptable accuracy losses while maximizing hardware efficiency.

- Latency vs. Throughput – Reducing latency while maintaining high throughput can be difficult. Solution: Using hls streams (FIFOs) for continuous data flow, matching the parallelization of the inner design. Carefully managing memory access by instantiating enough LUTs.

- Scalability and Adaptability – Ensuring that the design can scale for different workloads and FPGA resources. Solution: Using templates and parameterized hardware design where the number of synthesized modules can be adapted to system requirements dynamically.

- Hardware/Software Co-Design Complexity – Integrating custom hardware into an existing complex hardware design can introduce compatibility issues. Solution: Careful study of the hierarchy of functions in the network, and how

data is passed through each step, including data types and precisions. Performing co-simulation before deployment to ensure that the custom hardware interacts correctly with the software framework.

- Debugging and Verification – FPGA debugging is more challenging than software debugging due to limited observability. Solution: Utilizing simulation and co-simulation with its waveform viewer, to detect issues early.

4. The Accelerator

The final product of this thesis is a generalized LUT-based method of implementing activation functions, which can be used to accelerate different univariate functions. This chapter thoroughly describes the design process of this accelerator, starting from setting the goal to accelerate an activation function, up to having a final, flexible, custom hardware block, ready to be integrated into larger systems. The first activation function to be accelerated was the sigmoid, which generated the generalized LUT-based implementation. This implementation was then used to accelerate the SiLU activation function, attesting to the adaptable nature of this method.

4.1 Sigmoid

A logistic function or logistic curve is a common S-shaped curve with the equation,

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}} \tag{4.1}$$

where L, k and x_0 are tunable parameters.[25] Setting $L = 1$, $k = 1$ and $x_0 = 0$, we get the standard logistic function, otherwise known as the sigmoid, expressed with the equation,

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.2}$$

and depicted below in Figure 4.1. In equations 4.1 and 4.2, e is Euler's number (approximately 2.71828), and x is the input to the function. The sigmoid is a mathematical expression commonly used as an activation function in machine learning and neural networks. It maps any input to an output with a value in the range $[0, 1]$, making it a powerful solution when dealing with probabilities in classification problems.

A key point of interest for activation functions is their gradient and its computation. On one hand, the sigmoid is a smooth curve and has a smooth gradient,

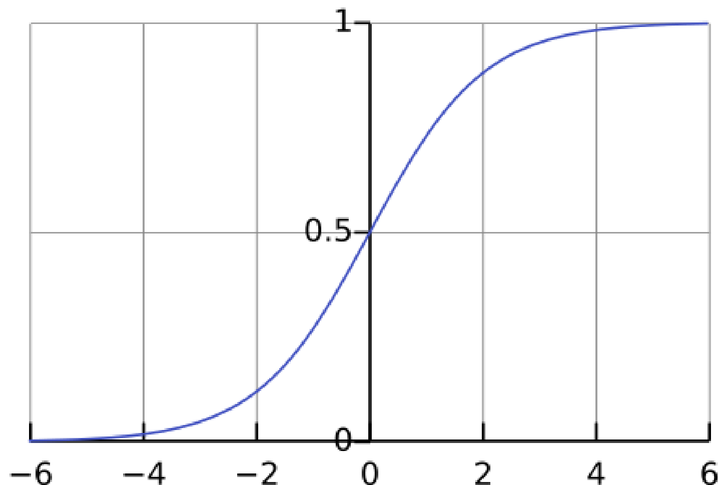

Figure 4.1: Standard logistic function[25]

meaning it is differentiable at all points, and thus ideal for the backpropagation process undergone during training of neural networks. On the other hand, this function suffers from the vanishing gradient problem, meaning that, when the input is very small or very large, the output saturates to 0 or 1, making the gradient converge to 0. Having an extremely small gradient is not ideal for backpropagation, slowing down or even stopping the training process for deep neural networks.

This thesis aims to tackle one of the biggest limitations this activation function has, which is how computationally expensive it is. Dealing with exponential functions and divisions can be very resource-intensive, needing for a lot of hardware to be allocated. A single hardware block doing sigmoid calculations could require hundreds of FFs, thousands of LUTs and many DSPs. For this reason alternative solutions to implement this function more efficiently needed to be researched.

4.1.1 Paper Review

This subsection provides a review of nine research papers that explore various approximation techniques, such as PWL functions, LUT method, and Taylor series expansion, that can be utilized for the sigmoid function. Each review will briefly describe the proposed method by the paper and highlight any key result and finding. This analysis serves to choose the most promising techniques to be tested as acceleration methods. Several other papers, such as *"A Novel Approximation Methodology and Its Efficient VLSI Implementation for the Sigmoid Function"*[26] and *"2b-sigmoid and 2b-tanh: Low Hardware Complexity Activation Functions for*

LSTM"[27], were reviewed, but for the sake of conciseness, only the most relevant papers have been discussed here.

Paper 1: Research and design of activation function hardware implementation methods[28]

- **Proposed Methods:** This paper serves as an introduction to the different methods, as it outlines some of the most popular approximation techniques for the sigmoid function. Implementations include piecewise linear function method, Taylor series expansion method and lookup table method.
- **Results & Findings:**
  - Look-up table method is the most universal. Within the allowable range of error, it stores sampling values of nonlinear functions. It can cost too much memory if not properly handled.
  - Taylor expansion method uses Taylor expansion formula (4.3) to simulate nonlinear function. The sigmoid is infinitely differentiable, so the Taylor expansion formula can theoretically approach any real value, but high order derivatives become too complex, and thus Taylor expansion method is rarely used.

$$f(x) = f(x_0) + f^{(1)}(x_0)(x - x_0) + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \tag{4.3}$$

  - Piecewise linear fitting method divides the nonlinear function, and uses multiple sections of linear functions to fit the nonlinear function within the allowable range of error (4.4).

$$\tilde{y} = \begin{cases} a_0 x + b_0, & x \in [x_0, x_0 + \Delta x] \\ a_1 x + b_1, & x \in [x_0 + \Delta x, x_0 + 2\Delta x] \\ \quad \vdots & \\ a_k x + b_k, & x \in [x_0 + k\Delta x, x_0 + (k+1)\Delta x] \\ a_{k+1} x + b_{k+1}, & x \in [x_0 + k\Delta x, x_0 + (k+2)\Delta x] \\ a_n x + b_n, & x \in [x_0 + (n-1)\Delta x, x_n] \end{cases} \tag{4.4}$$

Paper 2: Hardware Implementation of Sigmoid Activation Functions using FPGA[29]

- **Proposed Methods:** This paper shows 2 different implementations, a PWL one and an approximation with a 2nd degree polynomial.

  The first one, namely the PWL function method, divides the sigmoid into 4 intervals, computing only for the positive valued input, since the output for negative values can be computed as $f(-x) = 1 - f(x)$, due to the inherent symmetry of the sigmoid. The 4 intervals, described in the equation 4.5 and illustrated in Figure 4.2, divide the function into 8 overall sections of linear functions, using slopes and biases mainly in powers of two for efficiency. These exact 4 intervals and approximating functions were used in many implementations of other papers, such as *"Hardware Realization of Sigmoid and Hyperbolic Tangent Activation Functions"*[30]

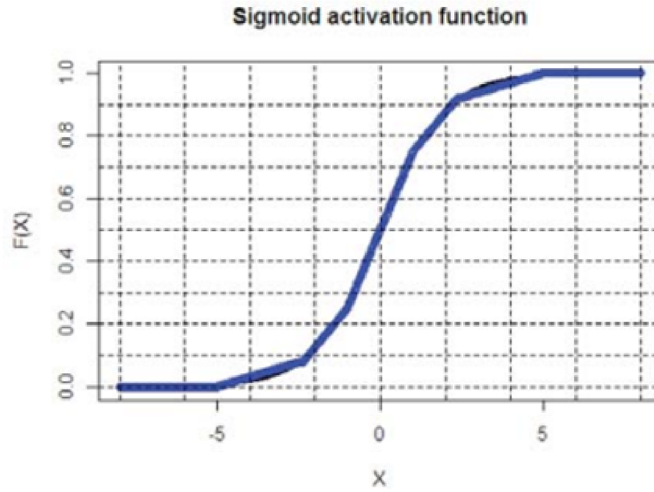

Figure 4.2: Sigmoid function and its PWL approximation

$$f(x) = \begin{cases} 1, & |x| \geq 5.0 \\ 0.03125 \cdot |x| + 0.84375, & 2.375 \leq |x| < 5.0 \\ 0.125 \cdot |x| + 0.625, & 1.0 \leq |x| < 2.375 \\ 0.25 \cdot |x| + 0.5, & 0 \leq x < 1.0 \end{cases} \quad (4.5)$$

The second method utilizes a quadratic polynomial expression (4.6) to approximate the sigmoid function, as seen in Figure 4.3. Again, the coefficients are powers of two to favor shifts instead of multiplications. An unoptimized approximation was also shown in the paper, with the coefficients not being powers of two. This provided slightly lower errors, but higher hardware cost for multiplications instead of shifts.

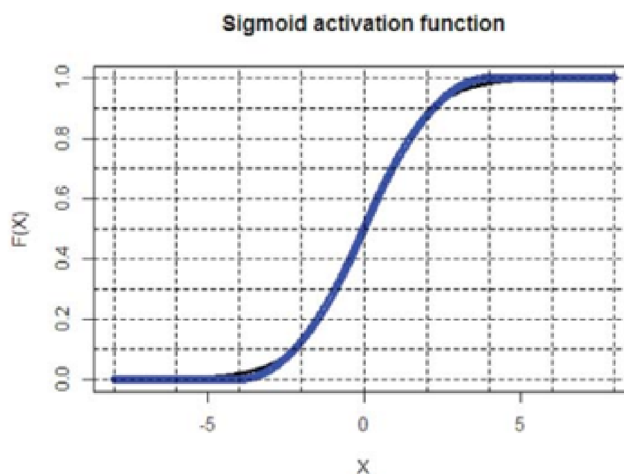

Figure 4.3: Sigmoid function and its polynomial approximation

$$f(x) = \begin{cases} 1, & x \geq 4.0 \\ -0.03125 \cdot x^2 + 0.25 \cdot x + 0.5, & 0 \leq x < 4.0 \end{cases} \tag{4.6}$$

- **Results & Findings:** The PWL method provides a lower cost solution, with errors to be further analyzed. The polynomial provides a lower error solution, with hardware cost to be assessed. Both of these approximation techniques are viable choices for implementation and further inspection.

Paper 3: FPGA Implementation for the Sigmoid with Piecewise Linear Fitting Method Based on Curvature Analysis[31]

- **Proposed Method:** This paper further studies the PWL method, by implementing different approximations with different numbers of intervals, starting from 5 up to 15. A graph containing the absolute errors of each implementation can be seen in Figure 4.4.

  After appropriate assessment, the paper proposes the approximation with 11 intervals as the best choice, as the errors stop decreasing substantially after that point. The PWL expression is given below (4.7), showing the 11 intervals.

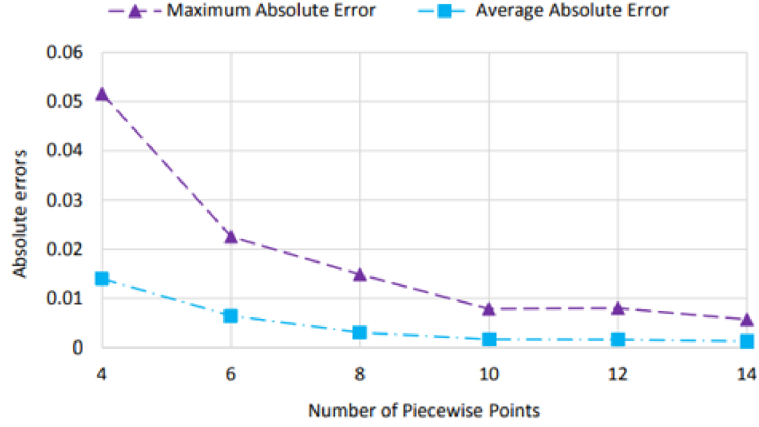

Figure 4.4: The absolute errors of different PWL function divisions

$$f(x) = \begin{cases} 0.00252 \cdot x + 0.01875, & x \leq -8, \\ 0.02367 \cdot x + 0.11397, & -8 < x \leq -4.5, \\ 0.06975 \cdot x + 0.25219, & -4.5 < x \leq -3, \\ 0.14841 \cdot x + 0.40951, & -3 < x \leq -2, \\ 0.2389 \cdot x + 0.5, & -2 < x \leq -1, \\ 0.14841 \cdot x + 0.59049, & -1 < x \leq 1, \\ 0.06975 \cdot x + 0.74781, & 1 < x \leq 2, \\ 0.02367 \cdot x + 0.88603, & 2 < x \leq 3, \\ 0.00252 \cdot x + 0.98125, & 3 < x \leq 4.5, \\ 1, & x > 8. \end{cases} \tag{4.7}$$

- **Results & Findings:** This PWL approximation provides higher accuracy than the one in the previous paper, however all of the slopes and biases not being powers of two, clearly increase the hardware requirements significantly, making this solution less viable than the one in Paper 2.

Paper 4: A Non-linear Approximation of the Sigmoid Function based on FPGA[32]

- **Proposed Method:** This paper presents another PWL function approximation approach, this time leveraging also non-linear functions in it. This method splits the sigmoid into 25 intervals, with some utilizing 2nd degree

polynomial expressions, while others linear ones, as depicted below in Figure 4.5.

TABLE I
THE INTERVAL/FUNCTION FORM

Interval	Function Form
x (-1,1)	y=0.2383x+0.5000
x [-2,-1]	y=0.0467* x^2 +0.1239*x+0.2969
x [-3,-2)	y=0.0298* x^2 +0.2202*x+0.4400
x [-4,-3)	y=0.0135* x^2 +0.1239*x+0.2969
x [-5,-4)	y=0.0054* x^2 +0.0597*x+0.1703
x [-5.03,-5)	y=0.0066
x [-5.2,-5.03)	y=0.0060
x [-5.41,-5.2)	y=0.0050
x [-5.66,-5.41)	y=0.0040
x [-6,-5.66)	y=0.0030
x [-6.53, -6)	y=0.0020
x [-7.6,-6.53)	y=0.0010
x [-∞,-7.6)	y→0
x [1,2)	y=-0.0467* x^2 +0.2896*x+0.4882
x [2,3)	y=-0.0298* x^2 +0.2202*x+0.5600
x [3,4)	y=-0.0135* x^2 +0.1239*x+0.7030
x [4,5)	y=-0.0054* x^2 +0.0597*x+0.8297
x [5,5.0218)	y=0.9930
x [5.0218,5.1890)	y=0.9940
x [5.1890,5.3890)	y=0.9950
x [5.3890,5.6380)	y=0.9960
x [5.6380,5.9700)	y=0.9970
x [5.9700,6.4700)	y=0.9980
x [6.4700,7.5500)	y=0.9990
x [7.5500,+∞)	y→1

Figure 4.5: PWL functions for the 25 intervals

- **Results & Findings:** This method provides the best approximation in terms of accuracy among the various PWL ones studied in these papers. However, this comes at an exorbitant cost, needing to compute numerous 2nd equations, requiring the allocation of the vast amount of resources required for multiplications and squares. The least viable of the PWL methods in terms of hardware.

Paper 5: High-Performance and Low-Cost Approximation of ANN Sigmoid Activation Functions on FPGAs[33]

- **Proposed Method:** This paper shows the last PWL method to be assessed in this thesis. After analyzing the implementation with highest hardware cost and best performance on the previous paper, this one implements the PWL function approximation with the lowest hardware cost possible and lowest performance. It is a PWL approximation using only 3 intervals, with 1 linear equation modeling the central part of the sigmoid, and 2 constant functions for either side of the sigmoid where it saturates (4.8).

$$f(x) = \begin{cases} 0, & x \leq -2, \\ x/4 + 0.5, & -2 < x \leq 2, \\ 1, & x > 2. \end{cases} \tag{4.8}$$

This approach is illustrated below with the graph in Figure 4.6 and a circuit implementation in Figure 4.7.

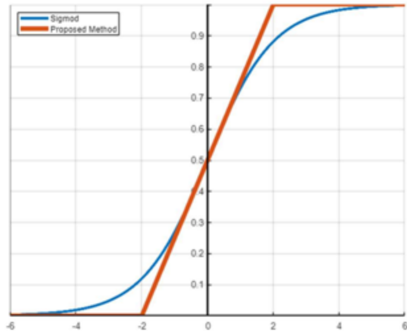

Figure 4.6: PWL approximation with 3 intervals

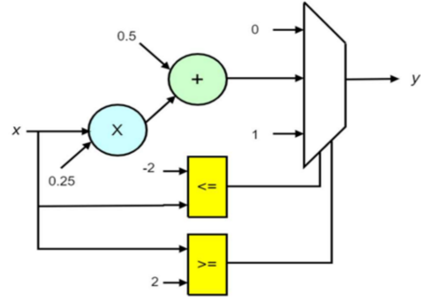

Figure 4.7: Proposed circuit implementation of sigmoid activation function

- **Results & Findings:** This approximation constitutes the lowest hardware cost amongst PWL implementations, while compromising on accuracy. Only one linear function needs to be computed, and the operations work on powers of two so for example the division by 4 will be a simple shift. The presented method is a viable solution to be tried in order to assess the error introduced by this approximation.

Paper 6: Design and FPGA Implementation of the LUT based Sigmoid Function for DNN Applications[34]

- **Proposed Method:** This paper proposes an implementation that uses an LUT based approach, where the initial LUT consists of 128 entries, each corresponding to a different sigmoid output value. This LUT needs 7 bits for the address. The hardware cost is them reduced by 50% considering the symmetry of the sigmoid, so we only need to use the first 64 entries of the LUT, as the others are computed as y=1-y. With only 64 entries in the LUT, 6 bits for the address are enough. The input value is quantized to 6 bits and used as an address for the LUT. The input range considered for the LUT is [-2,2] with a step size of 0.0314. Outside that range, output saturates to either 0 or 1. A small section of the table with the quantized samples is displayed below in Figure 4.9.

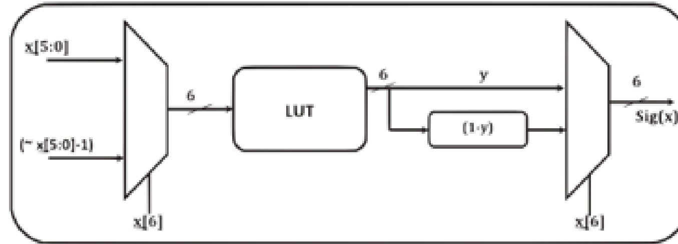

Figure 4.8: Block diagram of the proposed sigmoid hardware

Index	Address	$Prop.$ $Sig(x_n)$
0	0000000	0.1168
1	0000001	0.1201
2	0000010	0.1236
3	0000011	0.1271
4	0000100	0.1307
5	0000101	0.1344

Figure 4.9: First 6 quantized samples of the proposed LUT

- **Results & Findings:** This method displays excellent potential, if handled in a proper manner. The only operation computed is a subtraction, and this

is done to half the size of the table. This subtraction could be avoided by storing the full table. This would make for very low latency involving a simple memory read, low hardware cost due to no expensive operations, and the accuracy is configurable by tuning the coefficients stored in the table. Another viable solution for further study.

Paper 7: A Modular Approximation Methodology for Efficient Fixed-Point Hardware Implementation of the Sigmoid Function[35]

- **Proposed Method:** This paper describes how the sigmoid can be approximated using Taylor series expansion. The approach is made up of 3 modules: a PWL module, an EXP module and an NR module (Newton-Raphson), as illustrated in Figure 4.10.

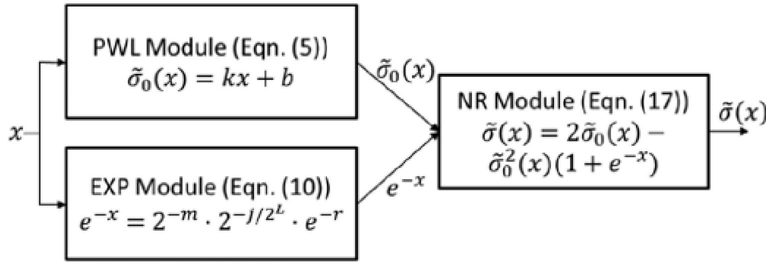

Figure 4.10: Workflow of modular approximation methodology NRA

The PWL module works exactly as the one presented in Paper 1.

The EXP module does some complex expression transformations to obtain the final expression (4.9),

$$e^{-x} = 2^{-m} \cdot 2^{-j/2^L} \cdot e^{-r} \tag{4.9}$$

and considering the 2 design choices: L=4 and Taylor Series Polynomial has only 1 leading term (1-r), (4.10) is obtained.

$$e^{-x} \approx 2^{-m} \cdot 2^{-j/16} \cdot (1 - r) \tag{4.10}$$

The NR module does an iterative process to approximate a function through tangent lines. The result at the end is computed using the 2 values obtained from the PWL and EXP modules, as can be seen in (4.11) and (4.12).

$$\tilde{\sigma}(x) = 2\tilde{\sigma}_0(x) - \tilde{\sigma}_0^2(x)(1 + e^{-x}) \tag{4.11}$$

$$\tilde{\sigma}(x) \approx 2\tilde{\sigma}_0(x) - \tilde{\sigma}_0^2(x)\left(1 + 2^{-m} \cdot 2^{-j/16} \cdot (1 - r)\right). \tag{4.12}$$

- **Results & Findings:** This is a very complex and resource-intensive implementation leveraging mathematical concepts such as Taylor series expansion, which may be viable in some specific occasions needing for high accuracy and a modular design.

Paper 8: A Novel Sigmoid Function Approximation Suitable For Neural Networks on FPGA[36]

- **Proposed Method:** This paper proposes 4 different implementations, however only the last one of them is of interest. This logarithmic technique replaces the division and exponential, with a logarithmic calculation of base 2 and few shift operations, substituting the multiplication by 2 and division by 8, as can be seen below in Figure 4.11.

For $x \geq 0$:

$$f(x) = \frac{\log_2(2x + 0.486)}{8} + 0.63$$

For $x < 0$:

$$f(x) = \frac{-\log_2(-0.5x + 0.008458)}{16} + 0.07$$

Figure 4.11: Logarithmic approximation of the sigmoid

The logarithmic expression is optimized and computed in a more efficient way, separating the exponent from the mantissa 'm', then approximating the log of the mantissa as shown in (4.13) below,

$$Log(m) = \begin{cases} m - 0.92, & 1 \leq x < 1.84, \\ 0.5 * m, & 1.84 \leq x < 2. \end{cases} \tag{4.13}$$

- **Results & Findings:** Substantially lower hardware cost than other 3 techniques in the same paper, but still considerably more costly than other methods. It only works on floating point values.

Paper 9: A Reconfigurable High-Precision and Energy-Efficient Circuit Design of Sigmoid, Tanh and Softmax Activation Functions[37]

- **Proposed Method:** This paper utilizes the Fast-Convergence CORDIC algorithm, which computes less iterations than CORDIC, to calculate the sigmoid output values. It is reconfigurable, so by changing some coefficients, the CORDIC hardware can compute exponential functions or divisions depending on the need. The Figure 4.12 depicted below, displays various parameters, with the main ones being the hardware cost of implementing each of the 3 activation functions individually using the conventional CORDIC algorithm, versus implementing only one FC CORDIC block for all 3 to share.

	Second Order [1]	Taylor-based [2]	Conventional CORDIC			FC CORDIC
Reconfigurable	✕	✕	✕			✓
Adjustable Precision	✕	✕	✓			✓
Function	Sigmoid/ Tanh	Softmax	Sigmoid	Tanh	Softmax	Sigmoid /Tanh /Softmax
LUT	158	1450	444	414	581.3	948.4
FF	46	1220	124	123	174.1	189.3
DSP	0	3	0	0	0	0
Max Freq (MHz)	333	N.R	126.6	118	142.9	108
Max Error (%)	0.78	N.R	0.54	0.41	N.A.	0.31[a] 0.30[b] N.A[c]
*Latency (ns)	9.99	N.A.	213.6	211.6	273.8	127[a] 123[b] 140[c]
*Dynamic Power (mW)	N.R	N.A.	12	12	95	16[a] 17[b] 168[c]
*Energy Efficiency (nJ/operation)	N.R	N.A.	2.65	2.54	26	2.08[a] 2.07[b] 23.5[c]

[a] Sigmoid [b] Tanh [c] Softmax

Figure 4.12: Comparison of different hardware designs

- **Results & Findings:** This implementation works best when a neural network uses 3 activation functions, namely Sigmoid, Tanh, and Softmax, because all 3 can share the same reconfigurable CORDIC, so the network will instantiate only 1 piece of hardware, without needing 3 CORDICs.

4.1.2 Implementations

After conducting a thorough paper review, the most promising approximation methods were identified and implemented on Vitis HLS. This allowed for performance and resource utilization comparison between designs and finally the choice of the definitive acceleration technique to be implemented.

This subsection explores the various approximations implemented in Vitis HLS, detailing the structure, the optimizations done, the applied pragmas, and the final results achieved in terms of both hardware and performance.

Sigmoid v0 - Original

The very first implementation is the original, ideal sigmoid function (4.14). This serves as an object of reference for the accelerated versions of this function, in order to assess the performance and hardware cost of the optimized implementations and be able to compare it to just allocating a hardware block computing the sigmoid.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.14}$$

This version has no optimizations or pragmas, due to the purpose it serves. Figure 4.13 displays the synthesis report of Vitis HLS regarding the original sigmoid function. Allocating one block able to handle one floating point computation at a time, requires 9 DSPs, 394 FFs, and 1058 LUTs. The estimated clock period is 7.248 ns, considering that the clock target was set to the default value of 10.00 ns, with uncertainty of 2.70 ns. With this clock period, this implementation would need 12 cycles to compute the sigmoid output value.

Summary Synthesis Report - sigmoid

Estimated Quality of Results

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	7.248 ns	2.70 ns

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
sigmoid	12	120.000	13	no	0	9	394	1058	0

Figure 4.13: Synthesis report of the original sigmoid function

Sigmoid v1 - PWL with 5 Intervals

The first approximation method implemented is the piecewise linear function method with 5 intervals (4.5) from Paper 2. The initial design contains a few if statements to differentiate between intervals, and then produces the output value by computing the needed absolute value calculation, multiplication, addition, and final subtraction. This unoptimized version generates the synthesis report seen in Figure 4.14. All results, except estimated clock period, are worse than the original sigmoid, so improvements are needed.

Figure 4.14: Synthesis report of sigmoid v1 with no optimizations

Various optimizations and directives were applied to this design to generate better end-results. Firstly, doing shifting operations instead of multiplications since all slopes are powers of two. Secondly, storing all slopes and biases in arrays, and instead of calculating the output inside the if statements, only selecting the needed coefficients from the arrays, and computing the calculation after these statements, avoiding doing many computations in parallel and then choosing the desired output. Thirdly, applying pragmas setting target initiation interval to 1, and max latency to 1. Lastly, using fixed point values instead of floating point ones. Figure 4.15 displays the synthesis report of the final version of this accelerator. The improvements are drastic in all aspects. From 30 DSPs to no DSPs at all, from 2106 FFs to 0 FFs, and from 2517 LUTs to 288 LUTs, all whilst both errors remained virtually the same as before optimizations.

Sigmoid v2 - PWL with 3 Intervals

This second implementation is an alternative PWL function approach, discussed in Paper 5, which divides the sigmoid in 3 intervals (4.8), with a central linear expression to be calculated and two saturating regions either side, as illustrated previously in Figure 4.6. The straightforward implementation, with no optimizations, generates the synthesis report depicted in Figure 4.16. It can be immediately seen that the results coincide with what was expected, in comparison with the previous PWL implementation. The hardware cost decreases, in exchange for increased errors.

Similar optimizations as the ones seen in sigmoid v1 can be applied to this design, namely shifting instead of multiplying, storing coefficients in arrays, using

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	5.094 ns	2.70 ns

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
sigmoid	0	0.0	1	yes	0	0	0	288	0

HW Interfaces

```
Error 998 is: 0.000346005
Error 999 is: 0.000340998
Avg error is: 0.00598065 and max error is: 0.0190431
```

Figure 4.15: Synthesis report of sigmoid v1 after optimizations

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	6.543 ns	2.70 ns

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
sigmoid	10	100.000	11	no	0	5	395	546	0

HW Interfaces

```
Error 998 is: 0.000346005
Error 999 is: 0.000340998
Avg error is: 0.0241021 and max error is: 0.119198
```

Figure 4.16: Synthesis report of sigmoid v2 with no optimizations

fixed point instead of floating point, and applying the directives for II and latency. These changes improve significantly the performance and resource efficiency of this accelerator, as can be assessed from the results in the synthesis report of the final version of this implementation exhibited below in Figure 4.17. The improvement in resource utilization is similar to the previous PWL, avoiding the use of DSPs and FFs in this design, and decreasing the number of LUTs from 546 to 68. The performance in terms of latency and estimated clock period also improved substantially. The principal trade-off of this accelerator is the loss in accuracy, so this design is viable in contexts where accuracy constraints are not critical.

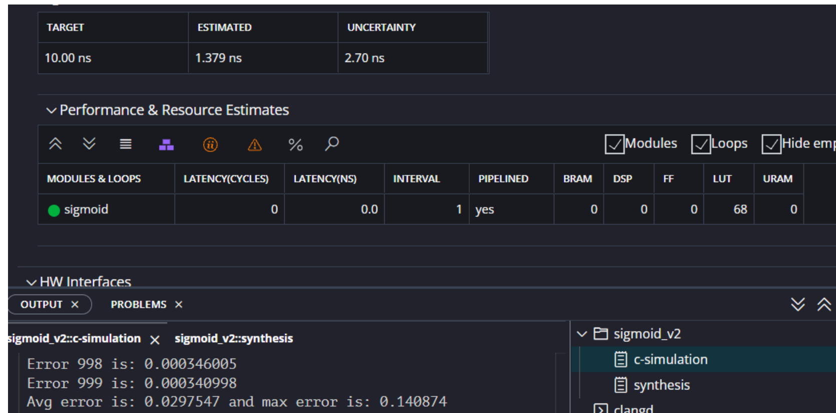

Figure 4.17: Synthesis report of sigmoid v2 after optimizations

Sigmoid v3 - 2nd Degree Polynomial

This accelerator is based on the second technique presented in Paper 2, approximating the sigmoid function with a 2nd degree polynomial equation (4.6). The initial design resulted in the metrics conveyed by the synthesis report in Figure 4.18.

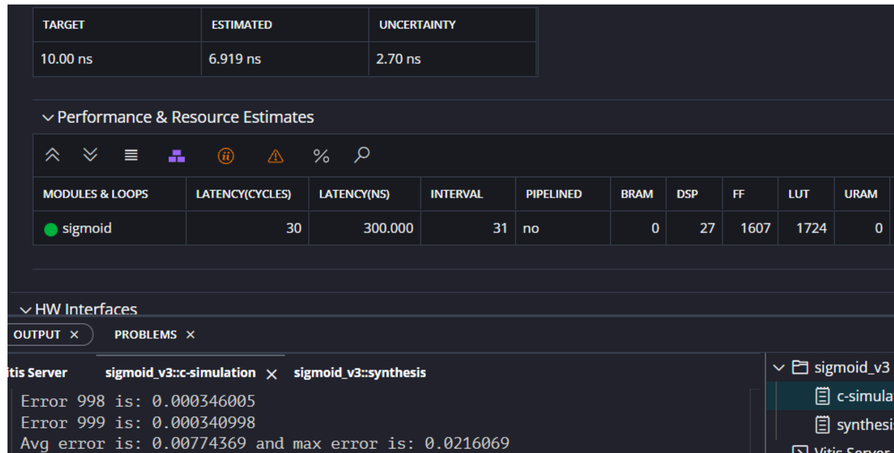

Figure 4.18: Synthesis report of sigmoid v3 with no optimizations

The enhancements done to this implementation are as follows: shifting instead of multiplying since slopes are powers of two, passing to fixed point from floating point, storing coefficients in arrays, and including the pragmas regarding II and max latency. The accelerator saw improvements in almost all aspects, as portrayed in Figure 4.19. It needs to be noted that this design still allocates one DSP module to calculate the square of the input. This can be avoided by writing a pragma in the

code, binding the specific multiplication operation to the fabric of the FPGA. In this way, instead of allocating 1 DSP and 147 LUTs, it will allocate 512 LUTs and no DSPs. This is a trade-off that can be taken into consideration in a case-by-case fashion, depending on the need and scarcity of DSPs in a specific larger design.

Figure 4.19: Synthesis report of sigmoid v3 after optimizations

Sigmoid v4 - LUT

This implementation is modeled after the LUT-based method described in Paper 6. A table containing quantized samples of the sigmoid function outputs is created and stored as an array. This accelerator receives an input value, manipulates it and transforms it into an address used to select an appropriate entry from the LUT, and returns this value as an output.

There are many degrees of freedom in this design, indicatively, the size of the LUT, datatype and bit-width of the stored coefficients, and the range represented by this table. The initial design, with degrees of freedom set to the parameter choices in Paper 6, and with the help of few minor refinements, such as shifting instead of multiplying to obtain correct address and using fixed point values, produced the metrics displayed in the synthesis report in Figure 4.20.

The design space to be explored was quite large, given the three degrees of freedom. The parameter choices that proved most of interest were: 128 and 256 entries for the size of the table, $[0, 4]$ and $[0, 8]$ for the range of the function, and finally 4 bits, 8 bits, and 16 bits, for the bit-width of the coefficients. For the sake of conciseness only the most relevant comparisons will be presented in this subsection, and more precisely LUT size of 128 vs 256 entries, quantized to 8 bits and in the range of $[0, 8]$. Figure 4.21 shows the juxtaposition between the

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	2.536 ns	2.70 ns

Performance & Resource Estimates

Modules Loops Hide er

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
sigmoid	1	10.000	1	yes	0	0	10	109	0

Figure 4.20: Synthesis report of sigmoid v4 after first optimizations

ideal sigmoid function and the 128 quantized samples of it in the range $[0, 8]$, that are stored in the look-up table. The 128 values for the negative input values of range $[-8, 0]$ are calculated at the end by computing y=1-y due to the symmetry of the function. On the other hand, Figure 4.22 illustrates the case where the LUT contains 256 entries instead of 128. Both of them model the original sigmoid function very well, making them viable solutions for an accelerator of the sigmoid.

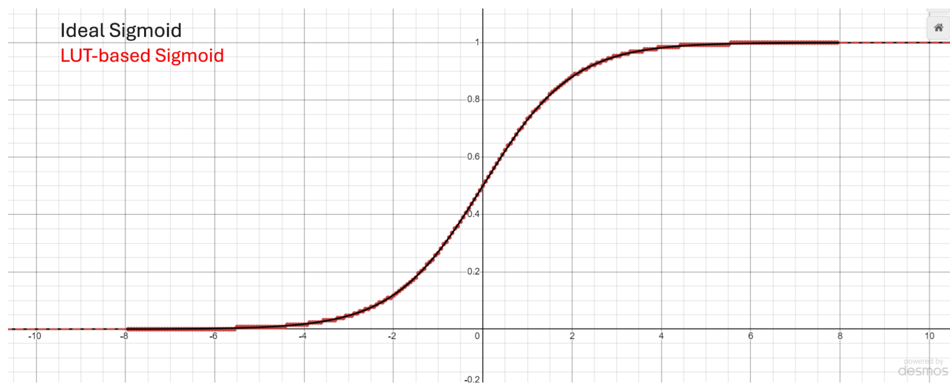

Figure 4.21: Sigmoid function approximation using a LUT with 128 entries of 8 bits with range $[0, 8]$

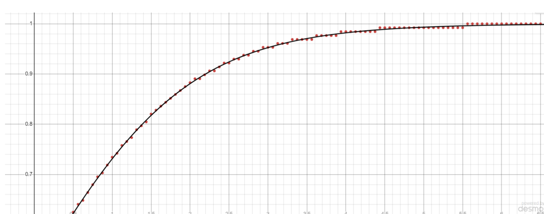

Figure 4.23: Zoomed in version of Figure 4.21

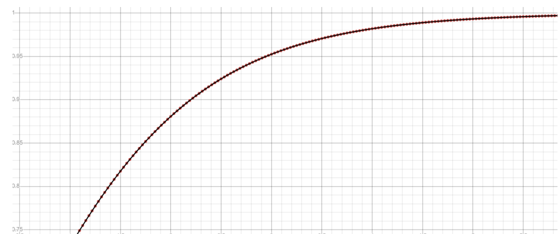

Figure 4.24: Zoomed in version of Figure 4.22

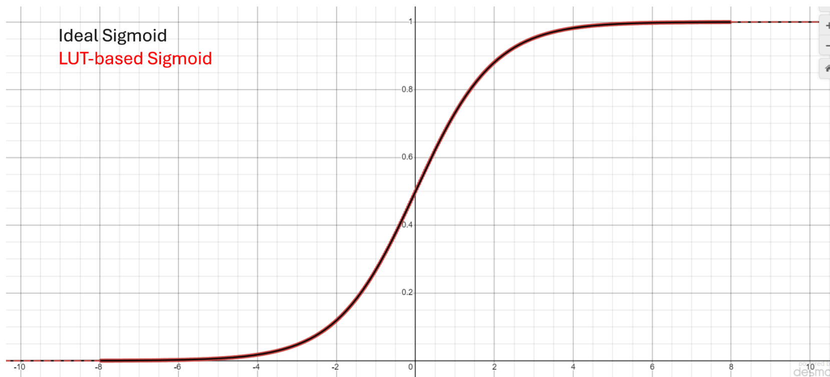

Figure 4.22: Sigmoid function approximation using a LUT with 256 entries of 8 bits with range $[0, 8]$

Sigmoid v5 - Final Version, LUT

This implementation is the final version of the sigmoid activation function accelerator, which also serves as the design for the generalized acceleration method mentioned earlier in this thesis. It is a highly improved design, based on the LUT-based approach implemented in sigmoid v4, and comprising of many new features, such as scalability, adaptability, and dynamically configurable precision.

This hardware block expects a fixed point value of 8 bits as an input, and returns the same datatype as an output. To generate the sigmoid output value, it uses the 8-bit input directly as an address for the LUTs containing 256 entries, so all the possible input values map one-to-one with the coefficients stored in the LUTs. The entries stored in the tables are quantized to 8 bits for hardware efficiency, but can be tuned depending on the context.

A fixed point variable of 8 bits can represent different ranges of values depending on the location of its fixed point, or in other words, its precision. The point can be placed next to any of the 8 bits, meaning 9 different possible precisions. Setting the point after the LSB means the range the variable would represent is $[-128, 127]$ if signed, or $[0, 255]$ if unsigned, with an LSB precision of 1. Moreover, placing the point in the middle of the 8 bits, would split the variable into 4 integer bits and 4 fractional bits, and represent a range of $[-8, 7.9375]$ if signed, or $[0, 15.9375]$ if unsigned, with an LSB precision of 0.0625. This variation in range depending on fixed point location means that using one LUT with 256 entries to approximate the sigmoid would not be a very effective choice, since it would only model a fixed range of the function. For this reason, the code includes 9 arrays of 256 elements each, acting as the 9 LUTs approximating the sigmoid in 9 different

ranges, so it can handle inputs with any precision. The innovative feature is the dynamically configurable precision, guaranteed by the fact that only one of these LUTs will be synthesized into actual RTL, depending on the input arriving to the block and its precision. It does so by using templated structs containing the arrays, so only one is always instantiated.

Another design choice was implementing the look-up tables using LUTRAM technology instead of BRAMs. This allowed for more BRAMs to be available to other resource-intensive blocks in a larger design, such as convolution layers that need to store large amounts of data as weights and biases, and BRAMs is the usual preferred choice for this.

The finalized product, when synthesized, generated the metrics depicted in Figure 4.25. The significant improvements with respect to all other accelerator versions, can be easily seen. It has the lowest hardware cost at 10 FFs, 34 LUTs, and 0 DSPs, and the highest performance in terms of estimated clock and thus potential throughput and latency. Implementing the look-up tables using BRAM technology instead of LUTRAM would give the resource utilization results estimated in Figure 4.26.

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	0.790 ns	2.70 ns

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
sigmoid	1	10.000	1	yes	0	0	10	34	0

Figure 4.25: Synthesis report of sigmoid v5 implemented in LUTRAM

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	1.352 ns	2.70 ns

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
sigmoid	1	10.000	1	yes	1	0	2	2	0

Figure 4.26: Synthesis report of sigmoid v5 implemented in BRAM

4.1.3 Results

This chapter explored various approaches to accelerating the sigmoid activation function, reviewing numerous methods from existing literature and assessing their feasibility for FPGA acceleration. Several design strategies were implemented and tested, each with trade-offs in terms of accuracy, resource utilization, and timing performances. Throughout this process, many optimization techniques were leveraged to enhance all metrics of the accelerators.

After extensive research and analysis, the best design was selected based on its balance of hardware cost, accuracy, and latency. The LUT-based accelerator described in the "sigmoid v5" implementation, provides a generalized method to accelerate different univariate activation functions, by simply changing the coefficients stored in the 9 look-up tables, to quantized samples of the target activation function. A practical example will be shown in the next section, using this method on the SiLU activation function. This optimized solution can also be easily integrated into real neural networks, as will be discussed in the following chapter.

4.2 SiLU

A Sigmoid Linear Unit function, also known as SiLU, is an activation function that smoothly transitions between linear and nonlinear behaviours. It is defined by the equation,

$$f(x) = x \cdot \sigma(x) \tag{4.15}$$

where $\sigma(x)$ represents the sigmoid function, given by

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.16}$$

Substituting the sigmoid into the SiLU equation, the following expression is obtained,

$$f(x) = \frac{x}{1 + e^{-x}} \tag{4.17}$$

This function is also known as the Swish activation function, and it has been shown to outperform traditional activation functions like ReLU in certain deep learning applications. Unlike ReLU, which abruptly sets negative inputs to zero, SiLU maintains a smooth curve, allowing small negative values to pass through instead of truncating them, as illustrated in Figure 4.27. A significant advantage of SiLU is its smooth gradient, which helps mitigate issues related to vanishing

gradients during backpropagation, similarly to the sigmoid function discussed previously. Since SiLU is differentiable everywhere and does not suffer from dead neurons like ReLU, it has gained popularity in modern deep neural network architectures.

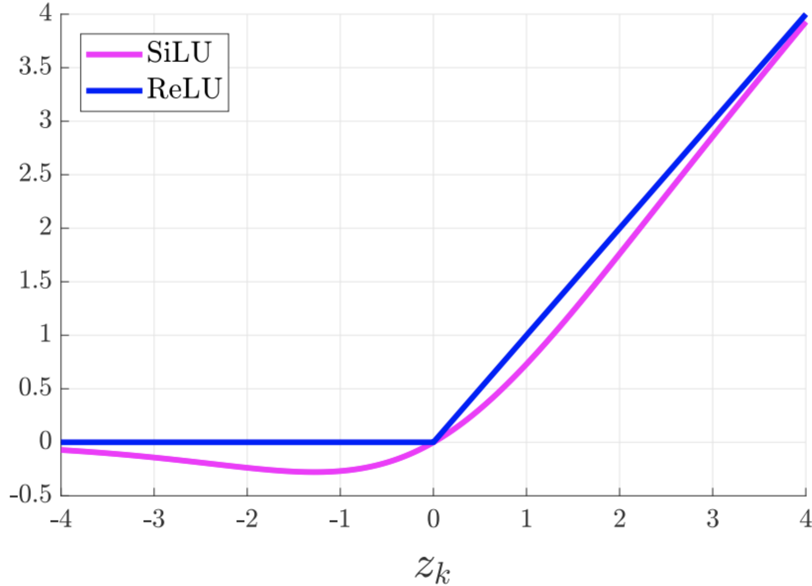

Figure 4.27: SiLU and ReLU activation functions[38]

Despite the many positive features it brings with it when utilized, it is burdened with some challenges, the likes of computational complexity. This activation function practically computes the sigmoid function and then multiplies the result by the input, adding a demanding operation to an already demanding function. As a result, accelerating the SiLU function is imperative in order to make it viable for real-time and resource-constrained applications, such as deep neural network inference on FPGAs. This section outlines how the generalized LUT-based acceleration method developed in this thesis can be applied to another univariate activation function, for instance the SiLU, conveying its adaptable nature.

4.2.1 Implementation

This accelerator was implemented using the LUT-based design developed for the sigmoid function. The only modification needed to be applied, was switching the coefficients in the LUTs from quantized samples of the sigmoid function to ones of the SiLU. These values were automatically generated by a tailored piece of code written for this purpose, streamlining the implementation process. Figure

4.28 below shows an example of the approximation of the SiLU function using the quantized samples of one of the LUTs, in the specific case where the expected input is an 8-bit fixed point value, with 5 integer bits and 3 precision bits, meaning the represented range of this input is $[-16, 15.875]$, with a precision of 0.125. The maximum error in absolute value of this approximation will never surpass the value of 1/2 of the LSB of the input, while the average error will typically be minuscule, assuming a randomly distributed tensor of data as input.

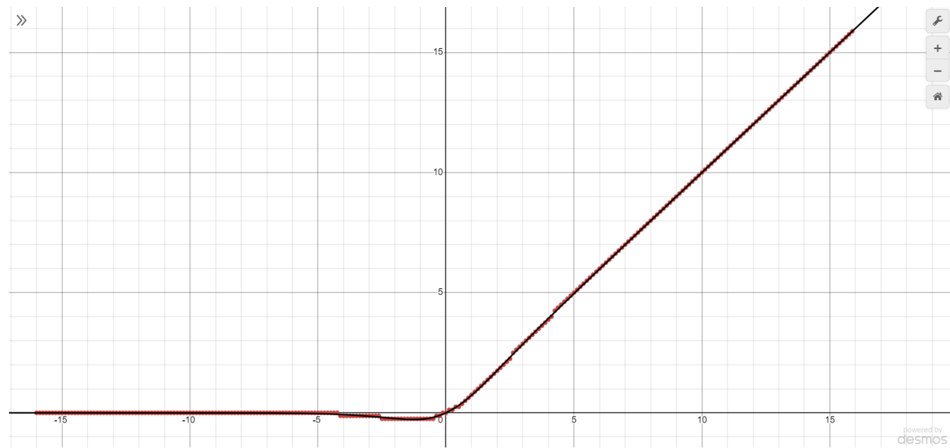

Figure 4.28: SiLU function approximation using a LUT with 256 entries of 8 bits for input range of [-16, 15.875]

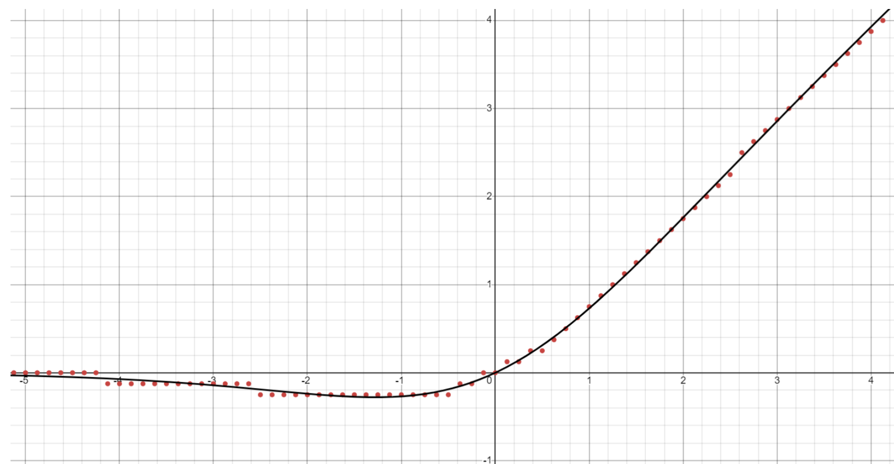

Figure 4.29: Zoomed in version of Figure 4.28

4.2.2 Results

Section 4.2 highlighted the practicality of the designed accelerator to be applied to a generic univariate activation function, which in this case was the SiLU. By simply modifying the values in the look-up tables, the same hardware design used for the sigmoid was able to accelerate this activation function. Thus, all the metrics, such as resource utilization and performance, for the accelerated SiLU, are the same as the ones for the sigmoid, and would still be the same if utilized for other activation functions like Tanh or GELU.

The decisions to accelerate the sigmoid and the SiLU specifically were not without reason, as the following chapter will further demonstrate. The SiLU function was chosen as the activation function to replace the traditional ReLU in deep neural networks, such as the ResNet architectures, in order to improve the performance of these AI models. However, simply implementing a layer that computes the SiLU function on 32-bit floating point values, to replace the ReLU one operating on 32-bit floating point, and deploying it on an FPGA board, is totally unfeasible, due to resource-constraints. That being said, the newly generated, and highly efficient SiLU accelerator on 8 bits, can now be integrated into deep learning architectures, and used to substitute the lower-performing ReLU activation function, while keeping the model still deployable on the FPGA board. This process is explored thoroughly in Chapter 5.

5. Use Case Scenario

This chapter will delve into the complete process of integrating an accelerated activation function into deep neural networks. By doing so, it will showcase the exposure to the full-stack experience of working on deep learning models and deploying them on boards for real-time inference. This full-stack approach includes processes such as: training DNNs to achieve the best possible performance, modifying the structure of these networks by adding new layers or integrating a custom-designed hardware block to replace the activation function being used, assessing various performance metrics through ONNX runtime, simulation, and co-simulation, and deploying the AI models on an FPGA board to run real-time inference on an image dataset.

Deep neural networks have become the go-to solution for many of today's cutting-edge applications, and in particular image classification and object detection problems. ResNet architectures are an excellent choice for handling these kinds of tasks, due to several key features they possess, such as residual connections. These models make for great use case scenarios to implement the designed accelerator, since they typically use ReLU functions as activation functions, and thus can be enhanced by integrating the accelerated SiLU. This process will be demonstrated on two ResNet models, namely Resnet-8 and Resnet-20, by thoroughly detailing in the following sections the full-stack workflow carried out. Other networks that could have served as use case scenarios for this thesis, are DNNs that implement the sigmoid activation function, such as YOLO, U-Net, or Faster R-CNN.

5.1 ResNet-8

5.1.1 Residual Block

ResNet-8 is a lightweight version of the ResNet architecture, consisting of 8 layers and implementing residual blocks, depicted in Figure 5.1, which enable the model to train very deep networks avoiding the risk of vanishing gradients. The ResNet residual block has a convolutional layer followed by a batch normalization

layer, a ReLU activation function, and a second convolutional layer paired with a batch normalization layer. The output of this section is added to the initial input which is passed through a residual connection, and lastly another ReLU activation function is applied to this sum. These residual connections allow the network to bypass multiple layers, facilitating the flow of information through the network and improving performance, especially when it comes to deep learning tasks like image classification. The skip connection may or may not feature a 1x1 convolution block, depending on the architecture, which transforms the input into the desired shape for the addition operation.

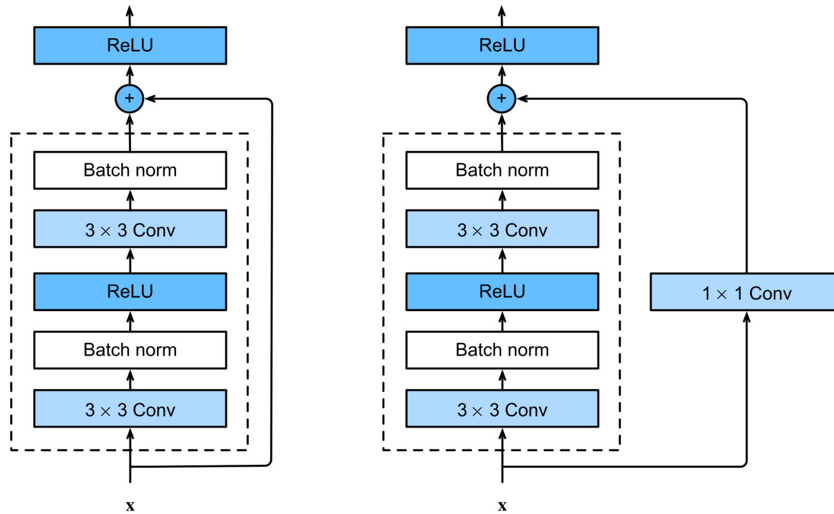

Figure 5.1: A ResNet residual block displaying a residual connection[39]

5.1.2 Modifying and Training the ResNet-8 Network

The work setup for this part of the research was briefly explained in Chapter 2, Toolchain and Technology. That chapter mentioned that VS Code was the IDE used to establish SSH connections to two remote machines, run Docker containers, and work on the codebase. The main codebase was pulled from the open source GitHub repository NN2FPGA[40], and a significant portion of time and effort was spent to study and understand the relevant source codes. The primary language utilized for training, modifying and running ONNX runtime was Python, working on the PyTorch framework.

Firstly, the original ResNet-8 model developed on Python was pulled and studied to deeply understand how layers are interconnected, what type of data is

passed to each of them and how it is processed. This step was crucial in order to be able to confidently modify and train this deep neural network. After this analysis, the model was modified, by replacing all the ReLU activation functions computing operations on 32-bit floating point values, with ideal SiLU activation functions working on 32-bit floating point values.

Secondly, quantization layers were added to this network, since the ReLU, previously implemented, was actually based on a QuantRelu module, meaning it automatically handled all the quantization issues for that layer, however the floating point SiLU did not provide the same feature. For this reason this feature had to be manually implemented by adding several QuantIdentity blocks to the model in all the needed locations, such as the inputs and outputs of layers, to guarantee correct quantization of values. The scaling implementation type of these blocks was set to "PARAMETER" to favor quantization-aware training of the network, enabling it to learn the scaling factor as a trainable parameter along the weights and biases during the training process.

Next, the modified network implementing the new activation function, needed to be trained since all parameters such as weights and biases were optimized for the ReLU layer. The design space for this training process depended on numerous elements, that can be divided into two main categories, hyperparameters and training algorithms. The hyperparameters tuned for best results were two, namely the learning rate and the batch size, varying them from 0.02 to 0.75 for the LR and from 16 to 256 for the batch size. On the other hand, the training algorithms explored were optimizers and schedulers, and the utilized ones were respectively SGD and CosineAnnealingLR.

The network was trained on classification tasks, leveraging the CIFAR-10 dataset. This dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The target device for this training was the powerful NVIDIA RTX 4090 GPU, which enabled very fast training times, while allowing for multiple trainings to take place concurrently. The results after training the network for 100 epochs can be visualized in Table 5.1. The best performing solution was selecting a batch size of 16, with a starting learning rate of 0.075, and with this setup the model was actually trained for 200 epochs instead of 100 for better convergence, resulting in an accuracy of 90.39%.

After training the model, and exporting its ONNX file, the following steps were to merge and fuse the convolution blocks with their subsequent batch normalization blocks, and retrain the modified network for 100 epochs, starting from the just

Table 5.1: Accuracy: Batch Size vs Learning Rate

BS\ LR	0.02	0.05	0.075	0.1	0.2	0.4	0.5	0.6	0.75
256	86.21	87.62	87.72	87.80	88.74	89.21	**89.4**	89.12	89.17
128	87.11	88.32	88.53	89.02	89.49	89.37	**89.6**	89.52	89.22
64	88.54	89.1	89.33	89.24	**89.55**	89.33	88.59	88.63	88.74
32	88.92	**89.44**	89.25	89.4	89.36	88.51	88.06	88.44	88.79
16	89.52	89.56	**90.39**	89.37	88.73	88.43	87.98	87.55	87.01

trained model and its saved parameters, stored as a checkpoint. The accuracy of the modified model decreased to 90.21%. A script was run to generate the final ONNX (QONNX) of this network, doing all the needed clean up and quantization of the model, and therefore creating the definitive model structure for the ResNet-8 architecture with SiLU FLOAT instead of ReLU FLOAT as activation functions. This final structure was tested by running inference on the 10000 test images of CIFAR-10 through ONNX runtime, and the resulting accuracy was 90.21% displaying congruency with what conveyed by the training results.

5.1.3 Integrating the Accelerator

The first step for integrating the accelerator into the trained ResNet-8 network utilizing the SiLU FLOAT activation function, is creating a custom layer on Python implementing the accelerated SiLU function, since the work is being done within PyTorch's framework. This layer will be a PyTorch module, which is a specialized form of a Python class, namely torch.nn.Module, designed to work within PyTorch's neural network framework. The accelerated SiLU will often be referred to as the SiLU LUT in this document.

In order to ensure correct functionality, the layer implemented on Python contains two structural changes with respect to the HLS code. Firstly, on Vitis HLS, the SiLU LUT block uses templates, so the input datatype can be passed to the block as a template parameter, and used directly to select the correct LUT to be instantiated. The same thing cannot be done in Python, so instead the complete input tensor is checked for the smallest positive, non-zero value, to infer the smallest input value, and thus the LSB, and leverage that to select the correct array. Secondly, in the high-level synthesis design, data is passed using 8-bit, fixed point variables, and these bits can be individually selected and manipulated to be used as an array index, however this is not possible in Python. To handle this issue, a scaling factor and a zero point bias are applied to the input, to generate the correct address for the LUTs.

With the new layer implemented on Python, the SiLU LUT is now ready to be integrated into the network. To do so the SiLU FLOAT layer in the Python code for the model is simply substituted by the SiLU LUT layer. The new model is exported to ONNX, then undergoes the same transformations as the previous model modification, indicatively batch normalization fuse, cleaning and quantization, to generate the definitive ONNX file for the model with the accelerated SiLU activation function. The network is then tested by running the same inference test as in the previous section, using the 10000 test images of CIFAR-10 dataset, leveraging ONNX runtime. The resulting accuracy was 90.21%, meaning no loss in accuracy for the model passing from SiLU FLOAT to SiLU LUT as activation function, while providing massive hardware cost reductions, and enabling the ResNet-8 network with SiLU as activation function, to be deployed and run on an FPGA board.

5.1.4 Synthesizing and Deploying the model on the Xilinx® Kria KV260 FPGA board

The final step in this full-stack workflow, was generating the full HLS code for the accelerated ResNet-8 network. To do so, a script provided in the codebase was used, which enabled the generation of full C++ source codes for neural networks starting from an ONNX. The HLS code for the accelerated SiLU still needed to be integrated into this large amount of source codes for the neural network. This was a complex task, since the codes were extensive, especially for heavy load layers such as convolution layers. For this reason a significant amount of time and effort was spent studying these source codes, understanding where to allocate the accelerated SiLU code, how to connect it appropriately to the vast amounts of data streams occurring in the system, handle datatypes properly, and last but not least how to exploit all the directives, optimizations, and parallelization strategies already implemented in the network.

After integrating the activation function's HLS code into the HLS code of the whole network and testing for functional correctness, the network was ready to undergo the Vitis HLS/Vivado synthesis workflow explained in Section 3.3. This was done by running a script present in the NN2FPGA framework, carrying out the complete workflow for both ResNet-8 versions, the original one with ReLU, and the one with the accelerated SiLU, in order to compare results. As shown in Table 5.5 below, there are some trade-offs in the allocated hardware resources, such as slightly less LUTs and BRAMs, more FFs and DSPs, and virtually identical critical path (CP) when synthesis target clock period is set to 5 ns. The resources allocated post-implementation were heavily influenced by the Integer Linear Programming algorithm utilized. ILP determines how instructions are efficiently computed

in parallel by taking into consideration objectives and constraints to optimize resources.

Model	kLUT	kFF	DSP	BRAM	URAM
ResNet-8 ReLU	66.9 (57.1%)	66.0 (28.2%)	768 (61.6%)	117 (40.6%)	0 (0.0%)
ResNet-8 SiLU	66.6 (56.9%)	72.7 (31.1%)	836 (67.0%)	109 (37.9%)	0 (0.0%)

Table 5.2: Resource utilization comparison between ResNet-8 with original ReLU and accelerated SiLU

The final products of these workflows were the bitstreams that were loaded onto the Xilinx® Kria KV260 FPGA board, effectively deploying the AI models on it, where real-time inference tests on the networks were carried out, resulting in an accuracy of 90.00% for the SiLU LUT, as can be seen in Table 5.6 below. The small difference in accuracy from ONNX runtime can be attributed to several possible factors, including quantization differences between the ONNX runtime environment and the FPGA board, and the effect of using floating point (ONNX runtime) or fixed point (FPGA) for the inference. Nevertheless, the final accuracy of the accelerated model is 90.00%, seeing an improvement of 1.02% from the original ResNet-8 model using the traditional ReLU, which displays an accuracy of 88.98%. The throughput is slightly improved by around 300 FPS, while power consumption increases very slightly and total time remains unchanged for a batch size of 10000.

Model	Dataset	Freq. (MHz)	Throughput (FPS)	Power (W)	Accuracy (%)
ResNet-8 ReLU	CIFAR10	200	29313	7.03	88.96
ResNet-8 SiLU	CIFAR10	200	29527	7.27	90

Table 5.3: Performance comparison between ResNet-8 with original ReLU and accelerated SiLU

5.2 ResNet-20

ResNet-20 is a more advanced version of the ResNet architecture compared to the ResNet-8. This network consists of 20 layers instead of 8, providing a deeper architecture able to perform deeper learning and achieve better inference results. As expected, this model is computationally more expensive due to its

larger complexity and size. For this reason, trying to implement traditional, non-optimized SiLU functions instead of ReLU functions, and being able to deploy it on the FPGA board, presents an even more unfeasible task than the ResNet-8, making the integration of the accelerated SiLU even more beneficial.

The full-stack workflow carried out for the ResNet-20 is the same as for the ResNet-8, only producing different results in values, and it is outlined below:

- Modifying the ResNet-20 model in Python by substituting the ReLU FLOAT activation function with the SiLU FLOAT one, and adding quantization layers to enable quantization-aware training.
- Training for maximum performance and accuracy, as illustrated in Table 5.4. Highest accuracy was 93.02% for selected learning rate of 0.075, and batch size of 16.

Table 5.4: Accuracy: Batch Size vs Learning Rate

BS\ LR	0.02	0.05	0.075	0.1	0.2	0.4	0.5	0.6	0.75
256	X	X	91.05	91.16	91.64	92.27	92.00	92.11	**92.58**
128	X	91.54	91.75	91.66	**92.43**	92.39	92.37	92.3	92.24
64	X	91.83	92.27	**92.73**	92.58	92.01	X	X	X
32	91.89	92.19	92.59	**92.78**	92.41	91.13	X	X	X
16	92.47	92.97	**93.02**	92.88	91.71	X	X	X	X
8	X	**92.63**	92.28	91.88	X	X	X	X	X

- Merging all convolution layers with their respective batch normalization layers and retraining, resulting in accuracy of 92.88%.
- Generating the final ResNet-20 ONNX model with SiLU FLOAT as activation function, after carrying out all the appropriate cleaning and quantization steps.
- Testing through ONNX runtime, using the 10000 test images of the CIFAR-10 dataset. Result was 92.88%, verifying correctness.
- Implementing the SiLU LUT layer in Python and modifying the model by replacing SiLU FLOAT with SiLU LUT.
- Generating the final ResNet-20 ONNX model with SiLU LUT as activation function, after carrying out all the appropriate cleaning and quantization steps.

- Testing through ONNX runtime, using the 10000 test images of the CIFAR-10 dataset. Result was 92.88%, guaranteeing again no loss in accuracy passing from SiLU FLOAT to SiLU LUT, as in ResNet-8.
- Generating the full network's HLS source code from the ONNX.
- Carrying out the whole Vitis-HLS/Vivado workflow for both ResNet-20 with ReLU FLOAT and SiLU LUT, producing the post-implementation resource usages depicted in Table below. A slight increase in all resources allocated can be noticed, with a minor improvement in CP. The ILP algorithm impact discussed for the ResNet-8 needs to be taken into consideration for this case as well.

Model	kLUT	kFF	DSP	BRAM	URAM
ResNet-20 ReLU	75.0 (64.0%)	71.5 (30.5%)	650 (52.1%)	147 (51.0%)	6 (9.4%)
ResNet-20 SiLU	77.6 (66.3%)	73.4 (31.4%)	668 (53.6%)	147 (51.0%)	6 (9.4%)

Table 5.5: Resource utilization comparison between ResNet-8 with original ReLU and accelerated SiLU

- Loading the bitstreams obtained from the previous workflow, onto the FPGA to effectively deploy the models.
- Running real-time inference, using the 10000 test images of the CIFAR-10 dataset. The result for SiLU LUT was 92.78%, showing a minimal decrease from ONNX runtime for the same reasons described in previous section.
- Comparing deployment accuracies of original model vs accelerated model. Original model showed an accuracy of 90.09%, while the accelerated one displayed an improvement of 2.69%, with a final deployment accuracy of 92.78%, as illustrated in the Table below. The throughput is slightly improved by around 31 FPS, while power consumption increases very slightly and total time remains unchanged for a batch size of 10000.

Model	Dataset	Freq. (MHz)	Throughput (FPS)	Power (W)	Accuracy (%)
ResNet-20 ReLU	CIFAR10	200	7550	6.35	90.09
ResNet-20 SiLU	CIFAR10	200	7581	6.90	92.78

Table 5.6: Performance comparison between ResNet-20 with original ReLU and accelerated SiLU

6. Conclusions

This thesis presented a generalized LUT-based method for accelerating univariate activation functions on FPGAs using High-Level Synthesis. This method was applied to two activation functions, namely the sigmoid and the SiLU, with the latter one being then integrated into two deep neural networks to assess the impact of the accelerated activation function on the deep learning models. The two networks, ResNet-8 and ResNet-20, were enhanced by replacing the traditional ReLU activation functions with SiLU ones, and then trained in a hardware-aware and quantization-aware manner. After substituting the SiLU functions with their accelerated counterparts, the trained models were deployed onto an FPGA, and their performances were analyzed in terms of accuracy, power consumption, latency and resource utilization.

The designed accelerator can be used in two different contexts. One is by replacing a resource-intensive activation function with its accelerated version, aiming to decrease computational complexity with tolerable accuracy loss, for instance accelerating the sigmoid function of a YOLO network. The other one is by replacing a traditional activation function with a more complex one, and then that one with the accelerated version of it, aiming for an improvement in accuracy with acceptable hardware trade-offs, like the discussed case of ReLU and SiLU in ResNet. These two cases are exhibited by the experimental results of this thesis. These results demonstrated that the LUT-based SiLU implementation, compared to the floating-point one, significantly reduced hardware cost while introducing no losses in accuracy. Moreover, compared to the floating-point ReLU implementation, the models saw an increase in accuracy with the SiLU LUT, while maintaining similar numbers in resource utilization, introducing minor trade-offs between LUTs and FFs.

On the other hand, certain limitations were observed. The developed LUT-based approach can only handle univariate activation functions. In order to handle functions depending on multiple input variables, such as Softmax, the look-up tables sizes would have to increase substantially, due to the exponential relationship between them. Moreover, this method is perfect for approximating univariate

functions working on 8-bit inputs. If the input is on 16 or 32 bits, a case-by-case study of the activation functions must be done. For instance, this method is an effective solution for the sigmoid function even if it is working on 32-bit, due to its saturating nature. However, a network working on 32-bits and using SiLU activation functions, would be out of the scope of this method, due to it's unbounded nature.

Future work could explore upon the products of this research by accelerating new activation functions using the developed method, integrating these accelerators in other more complex neural networks, or experimenting with other FPGA families. These possible developments would further validate the practical impact of this thesis on real-time AI applications, such as edge computing and embedded systems.

Bibliography

[1] Wikipedia contributors. *Machine Learning.* Accessed: 2025-03-01. 2025. URL: https://en.wikipedia.org/wiki/Machine_learning (cit. on p. ii).

[2] AWS. *What is a Neural Network?* Accessed: 2025-03-01. 2025. URL: https://aws.amazon.com/what-is/neural-network/#:~:text=A%20neural%20network%20is%20a,that%20resembles%20the%20human%20brain. (cit. on p. ii).

[3] SAS Institute. *Artificial Intelligence (AI): What it is and why it matters.* Accessed: 2025-03-01. 2025. URL: https://www.sas.com/en_us/insights/analytics/what-is-artificial-intelligence.html (cit. on pp. 1, 2).

[4] Wikipedia contributors. *Artificial Intelligence.* Accessed: 2025-03-01. 2025. URL: https://en.wikipedia.org/wiki/Artificial_intelligence (cit. on p. 1).

[5] A. M. TURING. «I.—COMPUTING MACHINERY AND INTELLIGENCE». In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/61209000/mind_lix_236_433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433 (cit. on p. 1).

[6] Kirti Vashee. *Artificial Intelligence: And You, How Will You Raise Your AI?* Accessed: 2025-03-01. 2017. URL: https://kv-emptypages.blogspot.com/2017/12/artificial-intelligence-and-you-how.html (cit. on p. 2).

[7] Indupama Herath. «Multivariate Regression using Neural Networks and Sums of Separable Functions». PhD thesis. Apr. 2022 (cit. on p. 3).

[8] Datacamp. *Introduction to Activation Functions in Neural Networks.* Accessed: 2025-03-01. 2025. URL: https://www.datacamp.com/tutorial/introduction-to-activation-functions-in-neural-networks (cit. on p. 4).

[9] Daily Dose of Data Science. *How to Reliably Improve Probabilistic Multiclass-classification Models.* Accessed: 2025-03-01. 2024. URL: https://blog.dailydoseofds.com/p/how-to-reliably-improve-probabilistic (cit. on p. 5).

[10] Siemens. *Working smarter, not harder: NVIDIA closes design complexity gap with HLS*. Accessed: 2025-03-01. 2025. URL: https://resources.sw.siemens.com/en-US/white-paper-working-smarter-not-harder-nvidia-closes-design-complexity-gap-with-hls/ (cit. on p. 6).

[11] Logic Fruit Technologies. *FPGA Design, Architecture and Applications*. Accessed: 2025-03-01. 2025. URL: https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/ (cit. on p. 6).

[12] Hackster. *Edge Computing- KV260 Vision AI Starter Kit*. Accessed: 2025-03-01. 2025. URL: https://www.hackster.io/prithvi-mattur/edge-computing-kv260-vision-ai-starter-kit-877e89 (cit. on p. 7).

[13] AMD. *AMD Vitis™ HLS for Intuitive Design and Productivity*. Accessed: 2025-03-01. 2025. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html (cit. on p. 13).

[14] learnpytorch. *Zero to Mastery Learn PyTorch for Deep Learning*. Accessed: 2025-03-01. 2025. URL: https://www.learnpytorch.io/01_pytorch_workflow/ (cit. on p. 16).

[15] design-reuse. *Boosting Model Interoperability and Efficiency with the ONNX framework*. Accessed: 2025-03-01. 2024. URL: https://www.design-reuse.com/articles/54536/boosting-model-interoperability-and-efficiency-with-the-onnx-framework.html (cit. on p. 17).

[16] hwupgrade. *NVIDIA GeForce RTX 4090*. Accessed: 2025-03-01. 2025. URL: https://www.hwupgrade.it/schede-tecniche/schede/nvidia/geforce-rtx-4090 (cit. on p. 17).

[17] Mario Casu Luciano Lavagno. *Embedded Electronic Systems for Artificial Intelligence and Machine Learning - "Platforms"*. Lecture slides, Politecnico di Torino. 2023 (cit. on p. 18).

[18] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. 2017. arXiv: 1703.09039 [cs.CV]. URL: https://arxiv.org/abs/1703.09039 (cit. on p. 21).

[19] Vladimír Kunc and Jiří Kléma. *Three Decades of Activations: A Comprehensive Survey of 400 Activation Functions for Neural Networks*. 2024. arXiv: 2402.09092 [cs.LG]. URL: https://arxiv.org/abs/2402.09092 (cit. on p. 21).

[20] A Vaisnav, Sandhya Ashok, Shatharajupally Vinaykumar, and R. Thilagavathy. «FPGA Implementation and Comparison of Sigmoid and Hyperbolic Tangent Activation Functions in an Artificial Neural Network». In: *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*. 2022, pp. 1–4. DOI: 10.1109/ICECET55527.2022.9873085 (cit. on p. 21).

[21] Kai Qian, Yinqiu Liu, Zexu Zhang, and Kun Wang. «Efficient Implementation of Activation Function on FPGA for Accelerating Neural Networks». In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181406 (cit. on p. 21).

[22] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. *Parallel Programming for FPGAs*. 2018. arXiv: 1805.03648 [cs.AR]. URL: https://arxiv.org/abs/1805.03648 (cit. on p. 21).

[23] *Vitis High-Level Synthesis User Guide, UG1399 (v2023.2)*. AMD. December 18, 2023 (cit. on p. 21).

[24] *Zynq UltraScale+ MPSoC Data Sheet: Overview, DS891 (v1.10)*. Xilinx, AMD. November 7, 2022 (cit. on p. 21).

[25] Wikipedia contributors. *Logistic function*. Accessed: 2025-03-10. 2025. URL: https://en.wikipedia.org/wiki/Logistic_function (cit. on pp. 27, 28).

[26] Zidi Qin, Yuou Qiu, Huaqing Sun, Zhonghai Lu, Zhongfeng Wang, Qinghong Shen, and Hongbing Pan. «A Novel Approximation Methodology and Its Efficient VLSI Implementation for the Sigmoid Function». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.12 (2020), pp. 3422–3426. DOI: 10.1109/TCSII.2020.2999458 (cit. on p. 28).

[27] Yuan Zhang, Lele Peng, Lianghua Quan, Shubin Zheng, Qiufeng Feng, Yonggang Zhang, and Hui Chen. «2b-sigmoid and 2b-tanh: Low Hardware Complexity Activation Functions for LSTM». In: *2022 19th International SoC Design Conference (ISOCC)*. 2022, pp. 93–94. DOI: 10.1109/ISOCC56007.2022.10031500 (cit. on p. 29).

[28] Chen Zhengbo, Tong Lei, and Chen Zuoning. «Research and design of activation function hardware implementation methods». In: *Journal of Physics: Conference Series* 1684.1 (Nov. 2020), p. 012111. DOI: 10.1088/1742-6596/1684/1/012111. URL: https://dx.doi.org/10.1088/1742-6596/1684/1/012111 (cit. on p. 29).

[29] Ivan Tsmots, Oleksa Skorokhoda, and Vasyl Rabyk. «Hardware Implementation of Sigmoid Activation Functions using FPGA». In: *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*. 2019, pp. 34–38. DOI: 10.1109/CADSM.2019.8779253 (cit. on p. 29).

[30] Subhanjan Konwer, Maria Sojan, P Adeeb Kenz, Sooraj K Santhosh, Tresa Joseph, and T.S. Bindiya. «Hardware Realization of Sigmoid and Hyperbolic Tangent Activation Functions». In: *2022 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)*. 2022, pp. 84–89. DOI: 10.1109/IAICT55358.2022.9887382 (cit. on p. 30).

[31] Zerun Li, Yang Zhang, Bingcai Sui, Zuocheng Xing, and Qinglin Wang. «FPGA Implementation for the Sigmoid with Piecewise Linear Fitting Method Based on Curvature Analysis». In: *Electronics* 11.9 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11091365. URL: https://www.mdpi.com/2079-9292/11/9/1365 (cit. on p. 31).

[32] Zhenzhen Xie. «A non-linear approximation of the sigmoid function based on FPGA». In: *2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI)*. 2012, pp. 221–223. DOI: 10.1109/ICACI.2012.6463155 (cit. on p. 32).

[33] Kostantinos Tatas and Michalis Gemenaris. «High-Performance and Low-Cost Approximation of ANN Sigmoid Activation Functions on FPGAs». In: *2023 12th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 2023, pp. 1–4. DOI: 10.1109/MOCAST57943.2023.10176636 (cit. on p. 34).

[34] Revathi Pogiri, Samit Ari, and K K Mahapatra. «Design and FPGA Implementation of the LUT based Sigmoid Function for DNN Applications». In: *2022 IEEE International Symposium on Smart Electronic Systems (iSES)*. 2022, pp. 410–413. DOI: 10.1109/iSES54909.2022.00090 (cit. on p. 35).

[35] Zhe Pan, Zonghua Gu, Xiaohong Jiang, Guoquan Zhu, and De Ma. «A Modular Approximation Methodology for Efficient Fixed-Point Hardware Implementation of the Sigmoid Function». In: *IEEE Transactions on Industrial Electronics* 69.10 (2022), pp. 10694–10703. DOI: 10.1109/TIE.2022.3146573 (cit. on p. 36).

[36] Peter W. Zaki, Ahmed M. Hashem, Emad A. Fahim, Mostafa A. Mansour, Sarah M. ElGenk, Maggie Mashaly, and Samar M. Ismail. «A Novel Sigmoid Function Approximation Suitable for Neural Networks on FPGA». In: *2019 15th International Computer Engineering Conference (ICENCO)*. 2019, pp. 95–99. DOI: 10.1109/ICENCO48310.2019.9027479 (cit. on p. 37).

[37] Bochang Wang, Ziang Duan, Zixuan Shen, Yuansheng Zhao, Lu Gao, and Chao Wang. «A Reconfigurable High-Precision and Energy-Efficient Circuit Design of Sigmoid, Tanh and Softmax Activation Functions». In: *2023 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. 2023, pp. 118–119. DOI: 10.1109/ICTA60488.2023.10364285 (cit. on p. 37).

[38] paperswithcode. *Sigmoid Linear Unit*. Accessed: 2025-03-13. 2025. URL: `https://paperswithcode.com/method/silu` (cit. on p. 48).

[39] d2l. *Residual Networks (ResNet) and ResNeXt*. Accessed: 2025-03-14. 2025. URL: `https://d2l.ai/chapter_convolutional-modern/resnet.html` (cit. on p. 52).

[40] Teodoro Urso Roberto Bosio. *NN2FPGA*. Accessed: 2025-03-14. 2025. URL: `https://github.com/robertoBosio/NN2FPGA` (cit. on p. 52).