INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Real-Time Bird Audio Detection using AI on FPGAs

Rodrigo Lopes da Silva

Licenciado

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor Mário Pereira Véstias
Doutor Rui António Policarpo Duarte

Júri:

Presidente: Doutor Tiago Miguel Braga da Silva Dias

Vogais: Doutor José Teixeira de Sousa
Doutor Rui António Policarpo Duarte

Julho, 2024

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Real-Time Bird Audio Detection using AI on FPGAs

Rodrigo Lopes da Silva

Licenciado

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor Mário Pereira Véstias
Doutor Rui António Policarpo Duarte

Júri:

Presidente: Doutor Tiago Miguel Braga da Silva Dias

Vogais: Doutor José Teixeira de Sousa
Doutor Rui António Policarpo Duarte

Julho, 2024

Acknowledgments

As I conclude this final chapter of my academic journey, I am deeply grateful for the support and encouragement that have enriched my experience.

I want to thank Professor Mário Véstias and Professor Rui Duarte for their insightful guidance and support throughout the development of this project.

I would also like to express my gratitude to other Professors from ISEL who had a positive impact on my learning experience. Professor Pedro Sampaio's experience in embedded systems sparked my interest in the field. Professor Nuno Cota's captivating classes were truly memorable, and I am grateful for his engaging teaching style. In addition, I want to thank Professor Pedro Pereira for the first programming lessons that fueled my passion for the subject.

Furthermore, I want to thank my family for their support over the years, for believing in my capabilities, and for helping me achieve my goals.

Lastly, I wish to extend my appreciation to my friends for providing a valued interruption to the usual routine, thus contributing to a life that is more balanced and enhanced.

Abstract

Audio-based monitoring offers a discreet solution for studying biodiversity behavior in remote or sensitive environments like forests. This work addresses the need for efficient wildlife monitoring, focusing on avian species using audio detection. This work optimizes one of the models related to the Bird Audio Detection Challenge (BADC), designs a hardware accelerator for the algorithm, and implements it in a System-on-Chip Field Programmable Gate Array (Xilinx Zynq UltraScale+ ZU3CG SoC). The model weights and activations are quantized and fine-tuned to improve the hardware performance and reduce resource usage without sacrificing much accuracy. The accelerator has different levels of quantization, 4 bits for the Convolution layers and 8 bits for the Gated Recurrent Unit (GRU) layers, implemented in the FPGA and integrated with the processor of the SoC-FPGA. The results show that the system has an accuracy of 79.5%, with reduced accuracy compared to the software Python model (89.75%). Still, it is acceptable since the objective is to reduce the model, implement it in hardware, and target 1 second or less evaluation time. The evaluation performance has a latency of 679ms, fulfilling the target delay of 1s. This work uniquely demonstrates the process of selecting a model, quantizing it, replicating the Python model in C, and implementing it into an FPGA. This represents a new project approach for a bird audio detection system within the scope of the BADC.

Keywords: Bird Audio Detection; Bird Audio Detection Challenge; Convolutional Neural Network; Recurrent Neural Network; Gated Recurrent Unit; TensorFlow; QKeras; Quantization; FPGA; Hardware Accelerator; High-Level Synthesis;

Resumo

A monitorização baseada em áudio oferece uma solução discreta para estudar o comportamento da biodiversidade em ambientes remotos ou sensíveis, como florestas. Este trabalho aborda a necessidade de uma monitorização eficiente da vida selvagem, focando-se em espécies de aves através da deteção por áudio. Este trabalho otimiza um dos modelos relacionados com o Desafio de Deteção de Áudio de Pássaros (BADC), projeta um acelerador em hardware para o algoritmo e implementa-o num System-on-Chip Field Programmable Gate Array (Xilinx Zynq UltraScale+ ZU3CG SoC). Os pesos e ativações do modelo são quantizados e ajustados para melhorar o desempenho do hardware e reduzir o uso de recursos sem sacrificar muito a precisão. O acelerador tem diferentes níveis de quantização, 4 bits para as camadas de Convolução e 8 bits para as camadas Gated Recurrent Unit (GRU), implementadas na FPGA e integradas com o processador do SoC-FPGA. Os resultados revelam que o sistema possui uma precisão de 79,5%, sendo esta inferior à precisão do modelo em Python (89,75%). No entanto, é aceitável, uma vez que o objetivo é reduzir o modelo, implementá-lo em hardware e alcançar um tempo de avaliação de 1 segundo ou menos. O desempenho de avaliação tem uma latência de 679ms, cumprindo o objectivo de uma atraso maximo de 1s. Este trabalho demonstra de forma única todo o processo, desde a seleção de um modelo, quantização, replicação do modelo Python em C e implementação numa arquitetura hardware/software reconfigurável. Representa uma nova abordagem de projeto de um sistema para deteção de áudio de pássaros no âmbito do BADC.

Palavras-chave: Detecção de Áudio de Pássaros; Desafio de Detecção de Áudio de Pássaros; Rede Neural Convolucional; Rede Neural Recorrente; Gated Recurrent Unit; TensorFlow; QKeras; Quantização; FPGA; Acelerador de Hardware; Síntese de Alto Nível;

Contents

Contents xi

List of Figures xiii

List of Tables xv

List of Listings xvii

Acronyms xix

1 Introduction 1

1.1 Motivation 1
1.2 Objectives 2
1.3 Source Code Availability 2
1.4 Report Outline 3

2 Background 5

2.1 Bird Audio Detection 5
2.2 Convolutional Neural Network 8
2.3 Recurrent Neural Network 11
2.4 Model Optimization 14
2.5 Model Design in FPGA 16
2.6 Conclusion 17

3 Model Design and Optimization 19

3.1 Workflow from BADC to FPGA 19

3.2 Model Selection 21

3.3 Exploring Quantization Techniques 27

3.4 Training Microfaune_ai with QKeras 28

3.5 Weights Extraction 34

3.6 Conclusion 36

4 Embedded Software Microfaune Model 37

4.1 TensorFlow Layers Independent Implementation in Python 37

4.2 Conclusion 48

5 HW/SW System for Bird Audio Detection 49

5.1 HW/SW System Architecture 49

5.2 IP Blocks 50

5.3 Post-Processing IP Blocks 61

5.4 Hardware Design 63

5.5 Software Design 63

5.6 Conclusion 66

6 Accuracy & Performance Results 67

6.1 Model Accuracy 67

6.2 Performance and Resources 68

6.3 Conclusion 73

7 Conclusions and Future Work 75

References 77

List of Figures

2.1 Example of a spectrogram in use for Bird Audio Detection (BAD), taken from microfaune-ai [19]. 7
2.2 Architecture of LeNet-5, a CNN for digits recognition 0-9. [10] 8
2.3 Example of conv2D. 9
2.4 Example of dense layer followed by softmax activation. [24] 11
2.5 Overview of a simple Recurrent Neural Network (RNN) 12
2.6 Gated Recurrent Unit (GRU) overview [15]. 13
2.7 Simple bidirectional architecture overview. 14
2.8 Time Distributed using Dense layer overview. 15
2.9 Quantization example where floating-point values are mapped to 4 bit fixed-point values 15

3.1 Project workflow 20
3.2 Model: All-Conv-Net-for-Bird-Activity-Detection-Significance-of-Learned-Pooling (ACNet) Max Pool Variant (MPV) 22
3.3 Model: ACNet Learned Pool Variant (LPV) 22
3.4 Seed search results for MPV, 87.9%. 24
3.5 Seed search results for LPV, 86.1%. 25
3.6 Model: microfaune_ai 26
3.7 Seed search results for *microfaune* 27
3.8 Original microfaune_ai, accuracy training progression plot, 90.50%. . . . 31

3.9 CNN quantized to 4 bits and RNN 64 cells, accuracy training progression plot, 85.46%. . . . 31
3.10 CNN quantized to 4 bits and RNN 1 cell, accuracy training progression plot, 84.91%. . . . 32
3.11 CNN quantized to 2 bits and RNN 64 cells, accuracy training progression plot, 65.07%. . . . 32

4.1 Print of each value of the GRU for the simple model. . . . 44
4.2 Overview of TensorFlow GRUv2 single cell step function. . . . 45

5.1 Proposed system architecture. . . . 50
5.2 Conv3D IP overview 51
5.3 FPGA input padding visualization. . . . 53
5.4 Small example of input sent over AXI-Stream. . . . 54
5.5 Small example of the circular input buffer. . . . 54
5.6 Small example of output sent over AXI-Stream. . . . 55
5.7 BGRU IP overview 57
5.8 Small example of an input sent in a Forward direction over AXI-Stream . 58
5.9 Small example of an input sent in a Backward direction over AXI-Stream . 58
5.10 Small example of output sent over AXI-Stream. . . . 60
5.11 Output conversion from Conv3D to BGRU, 4 bits to 8 bits. . . . 61
5.12 Example of BGRU post-process for GRUs with 2 cells. . . . 62
5.13 Vivado diagram. . . . 64
5.14 State diagram of the software that executes the microfaune_ai model. . . 65

6.1 Target Platform 69

List of Tables

3.1 Accuracy comparison between different amounts of bits used for quantization. . . . 33
3.2 Accuracy comparison between different amounts of GRU cells used, CNN quantized to 4 bits and RNN to 8 bits. . . . 34
3.3 Comparison of the number of parameters used by the Bidirectional GRUs with 64 cells vs 1 cell. . . . 34

5.1 Bytes used by Conv3D weights. . . . 53
5.2 Conv3D IP cycle unrolling cycles exploration. . . . 56
5.3 Bytes used by BGRU weights. . . . 59
5.4 BGRU IP expected FPGA resource usage. . . . 60

6.1 Accuracy between the Original, Modified, QKeras, and FPGA models. . . 68
6.2 Weights memory usage comparison between a model with 64 GRU cells and 1 GRU cell, using the previously mentioned weights calculations. . 68
6.3 Comparison of the milliseconds used in the CNN portion of the model, between Baseline and Optimized. . . . 70
6.4 FPGA resources comparison used in the CNN portion of the model, between baseline and optimized. . . . 70
6.5 Results of the execution time of both Bidirectional GRUs. . . . 71
6.6 FPGA resources used in the RNN portion of the model. . . . 71
6.7 Execution times taken on pre-processing between CNN, RNN, and final layers. . . . 71

6.8 Execution times of the final layers running on CPU. 72
6.9 Comparison of the number of milliseconds used in different sections of the model, between Baseline and Optimized. 72
6.10 Presenting the percentage of execution time across the multiple sections of the Optimized implementation. 72
6.11 Optimized resource usage. 73
6.12 Comparison of the number of milliseconds used in different sections of the model, between ARM CPU and Optimized. 73

List of Listings

3.1 Model setting used for training. 23
3.2 Changes to the original code adding support for setting a seed. 23
3.3 Added ReLu activation to quantize the input of the first Conv2D. 29
3.4 Conv2D and BatchNormalization replaced by QConv2DBatchnorm. . . 29
3.5 ReLu activation replaced by QActivation configured as ReLu. 29
3.6 Possible Bidirectional GRU replacement with QBidirectional and QGRU. 30
3.7 Function mergeKernelScale. 35
3.8 Function createScaleHLS. 35
3.9 Example of extracting weights from the 1st QConv2DBatchnorm. 35
4.1 Keras GRUv2 Python implementation. 43
4.2 Printing tensor values using print_tensor. 44
5.1 RNN output to Time-Distributed Dense input 62

Acronyms

ACNet All-Conv-Net-for-Bird-Activity-Detection-Significance-of-Learned-Pooling.

BAD Bird Audio Detection.

BADC Bird Audio Detection Challenge.

CNN Convolution Neural Network.

CPU Central Processing Unit.

DCASE Detection and Classification of Acoustic Scenes and Events.

DMA Direct Memory Access.

FM Feature Map.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

GRU Gated Recurrent Unit.

HLS High-Level Synthesis.

IP Intellectual Property.

LPV Learned Pool Variant.

LSTM Long Short-Term Memory.

MPV Max Pool Variant.

RNN Recurrent Neural Network.

SoC System-on-a-Chip.

Introduction

This project aims to adapt an existing Bird Audio Detection (BAD) model and optimize its performance by taking advantage of a System-on-a-Chip (SoC) with an Field Programmable Gate Array (FPGA) built-in. This chapter describes the motivation, the main objectives, and the report outline for this work.

1.1 Motivation

Birds offer an excellent example of animal behavior monitoring such as measuring biodiversity, population surveillance, and migratory pattern analysis, [2]. However, conducting visual surveys in remote or ecologically sensitive areas such as forests can be challenging and disruptive. In this situation, audio-based monitoring presents a discreet and effective solution, enabling researchers to gather data while minimizing disturbance to natural habitats. Sound-based wildlife detection lies in its ability to provide insights into the behaviors and trends of avian species.

Inspired by the recognition of the role that wildlife detection plays in ecological research, this project is motivated by the goal of enhancing the efficiency of a BAD model. Due to the computational workload demands intrinsic to neural networks, their execution commonly relies on a Central Processing Unit (CPU) or a Graphics Processing Unit (GPU). However, these choices come with drawbacks in power consumption, weight, and size.

A candidate solution is to use a centralized processing station for audio processing, with embedded systems collecting audio and sending the raw data to this station. However, this method generates a large amount of data for transmission, which is time and energy consuming.

An alternative solution is to use an all-in-one system, SoC-FPGA, that acquires and processes the audio locally and only sends the final evaluation result to the central station, resulting in a compact system with improved energy efficiency.

1.2 Objectives

The main objective of this work is to search, select, and adapt an existing BAD software model to be implemented in hardware using an SoC-FPGA. This implies studying the models found, primarily the ones in the Bird Audio Detection Challenge (BADC). The chosen model requires quantization, a step that could necessitate quantization-aware training, demanding the computational power of a GPU, while inference will be made in the FPGA. This quantization reduces the model size while minimizing its impact on accuracy. The hardware acceleration will be fine-tuned, validated, and compared to the original model's accuracy. Additionally, its performance will be benchmarked against a software implementation running on the ARM CPU of the embedded system, the same system containing the FPGA.

1.3 Source Code Availability

The source code for this project is available on GitHub. The repository contains all the necessary files, including the Python models, weights extraction scripts, C implementation, HLS implementation, and FPGA configurations.

You can access the repository using the GitHub link[1].

The datasets can be found at the DCASE Community website[2], under Development Datasets of Challenge2018, Task3.

And finally, the updated microfaune-ai model can be found in a forked GitHub repository[3].

[1] Bird Audio Detection Challenge - FPGA: https://github.com/PiniponSelvagem/BAD_FPGA
[2] Datasets: https://dcase.community/challenge2018/task-bird-audio-detection
[3] Updated microfaune-ai model: https://github.com/W-Alphonse/microfaune

1.4 Report Outline

This document is organized as follows. Chapter 2 introduces the background necessary to understand this work, such as the Bird Audio Detection Challenge, its datasets, and the layers used by the models explored. Chapter 3 explains the project workflow, model selection, quantization, and model weights extraction. Chapter 4 presents the model layers replicated from Python to C, preparing them for the hardware implementation. Chapter 5 describes the HW/SW system architecture, the IP blocks, hardware and software design. Chapter 6, the accuracy and performance results of the project. Lastly, chapter 7, the conclusions and future work.

2

Background

This chapter provides the background information necessary to understand this project, as well as the state-of-the-art related to Convolution Neural Network (CNN), Recurrent Neural Network (RNN), tools used for development, and Field Programmable Gate Array (FPGA) implementations of said neural networks.

2.1 Bird Audio Detection

Detecting birds from sounds is very useful for automatic wildlife monitoring. Detecting the presence of birds precedes other tasks, like bird counting or classification. In general, the process of Bird Audio Detection (BAD) can be split into two categories: simple bird sound detection, and bird species identification through their characteristic sounds. The first method simply detects any bird sound in an audio recording, helping to identify bird activity in an area. The second method focuses on recognizing different bird species by their unique sounds, providing a detailed view of the variety and numbers of birds in that area.

Algorithms for BAD have been extensively researched and developed, driven by the Bird Audio Detection Challenge (BADC) as part of the Detection and Classification of Acoustic Scenes and Events (DCASE) [7] that took place between 2016 and 2017, with a second edition in 2018. The main goal of this challenge was to advance the field of automatic bird audio detection and classification by developing machine learning models.

During the challenge, participants were provided with six datasets, with the task of generalizing their model to accurately evaluate the presence of bird audio vocalizations. Among these datasets, three were for development, while the remaining three were designed for evaluation. They contain 10-second-long WAV files at a sampling rate of 44,1 kHz mono PCM.

- Development datasets, binary labeled
  1. **Field recordings (freefield1010)** - a collection that contains 7690 recordings around the world of very diverse environments such as city sounds, wildlife, birds, etc. This dataset was gathered by the *Freesound* [12] project and then standardized for research.
  2. **Crowdsourced dataset (warblrb10k)** - this dataset comes from a UK bird-sound crowdsourcing research called *Warblr* [27]. It contains 8000 recordings from around the UK, by users using their smartphones using the Warblrb recognition app. The audio was gathered from multiple types of environments including weather noise, traffic noise, human speech, and even human bird imitations.
  3. **Remote monitoring flight calls (BirdVox-DCASE-20k)** - 20000 recordings near Ithaca, NY, USA during the autumn of 2015. It contains wildlife at night with around 50% of them containing at least one vocalization. [3] [4] [5]
- Evaluation datasets, non-labeled
  1. **Crowdsourced dataset (warblrb10k)** - same as the previously mentioned warblrb10k development dataset, but this version with only 2000 recordings.
  2. **Remote monitoring dataset, Chernobyl (Chernobyl)** - this dataset comes from the *Natural Environment Research Council (NERC)*, which has remote monitoring equipment in the Chernobyl Exclusion Zone. This dataset contains 6620 audio recordings covering a wide range of birds, other animals, and weather noise, gathered in environments such as abandoned villages, grassland, and forest areas.
  3. **Remote monitoring night-flight calls, Poland (PolandNFC)** - a collection of 4000 audio recordings from Hanna Pamula's Ph.D. project, focused on monitoring the nocturnal bird migration during the autumn season. This dataset only contains part of the original recordings. These recordings were

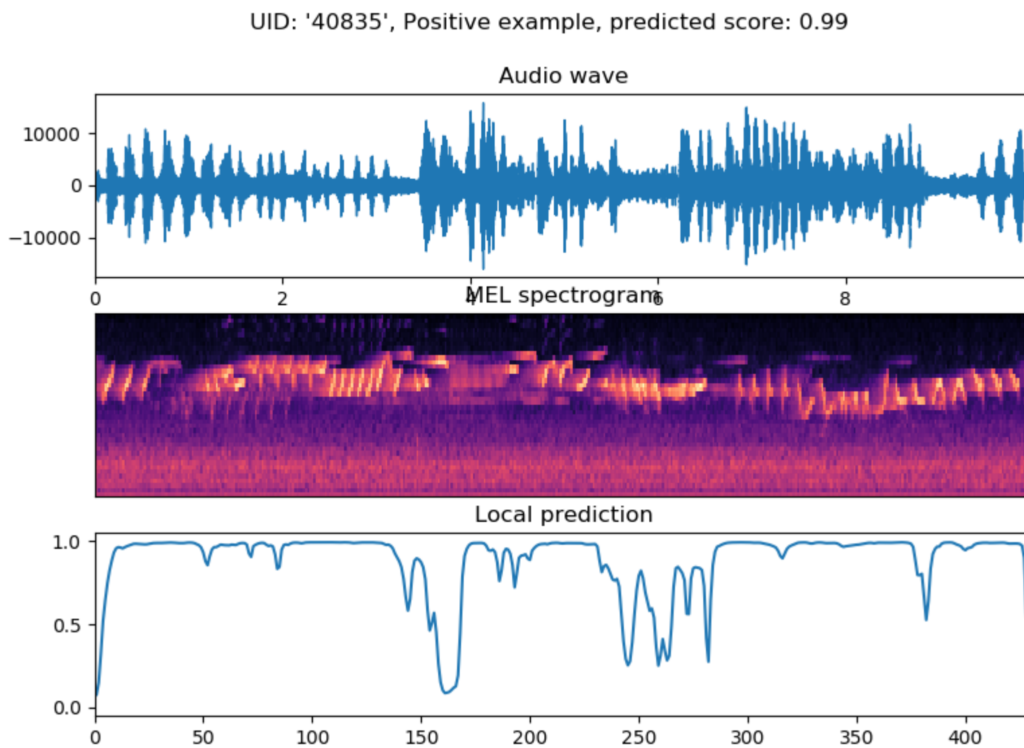


Figure 2.1: Example of a spectrogram in use for BAD, taken from microfaune-ai [19].

gathered each night from September to November 2016 along the Baltic Sea coast, of Poland. Contains recordings during different weather conditions and background noises, such as wind, rain, sea noise, insect calls, human voices, and even deer calls.

A bird audio detection system must process short audio recordings and return a binary decision, or a probability, about the presence or not of a bird sound.

With the recent advancement of deep neural networks, a very accurate solution extracts the spectrogram from the audio input 2.1 and sends it to a Convolution Neural Network (CNN) and/or Recurrent Neural Network (RNN) for feature extraction and classification. The spectrogram can be stored as an image, making it suitable for evaluation by a CNN.

An example of a spectrogram visualization alongside real bird vocalizations in action, can be observed in the video "Bird song Spectrogram" [1].

[1]Bird spectrogram visualization: https://www.youtube.com/watch?v=qp34UfXY_Wk

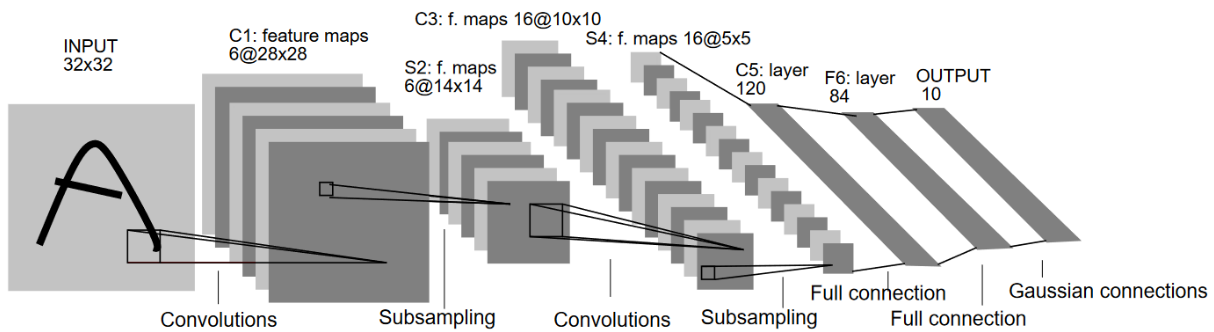


Figure 2.2: Architecture of LeNet-5, a CNN for digits recognition 0-9. [10]

2.2 Convolutional Neural Network

Convolution Neural Network (CNN) is a type of artificial neural network designed for processing grid-like data, such as images and videos. Both types can serve as inputs for the CNN, and the resulting output can be interpreted as a prediction, based on the model training. Model training involves using pre-existing classified datasets containing images and their corresponding classification labels. This process may extend over multiple iterations, refining the model's performance until it produces acceptable predictions.

CNNs are composed of several interconnected layers to extract features from input data. An early example is LeNet, dating back to 1998 (See Figure 2.2). It was designed to recognize characters and digits. LeNet includes convolutional layers, activation functions, pooling, and fully connected layers.

Convolutional layers detect local features such as edges and shapes, generating feature maps (Feature Map (FM)). Activation functions decide the importance of each output. Pooling layers reduce output size while preserving key details. Finally, fully connected layers serve as a classifier, combining learned features to make predictions.

2.2.1 Convolutional Layer

Convolutional layers (Conv2D) detect local features in input data by using kernels and bias. The kernel, a small two-dimensional array of trained weights, slides across the input data. The kernel performs a dot product at each position between its weights and the corresponding input data value. A bias associated with the kernel is added to the dot product result, adjusting the output FM's offset. Convolutional layers may also have parameters like stride and padding. Stride dictates how much the kernel moves

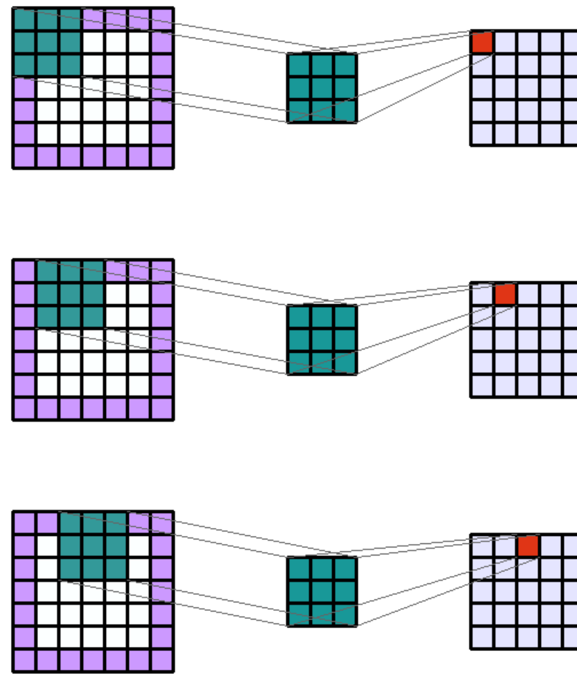


Figure 2.3: Example of conv2D.

with each step, while padding determines whether zero-valued positions surround the input data.

The output FM can be calculated using the equation 2.1 [6] where k_w, k_h are the width and height dimensions of the kernel, i_w, i_h are the width and height dimensions of the input, p is the size of padding, and s is the stride factor.

$$\text{output FM size} = \left(\left(\frac{i_\text{h} \cdot 2p_\text{h} - k_h}{s_h}\right) + 1\right) \cdot \left(\left(\frac{i_w \cdot 2p_w - k_w}{s_w}\right) + 1\right) \quad (2.1)$$

Figure.2.3, shows the first three steps of a simple convolution. In this example, the input data has a size of 5x5, and the conv2D parameters are a kernel 3x3 in dark green, with stride 1 and padding 1 at all sides in purple. In red is the output FM, although this example does not show bias, it would be added to at this last stage.

2.2.2 Batch Normalization Layer

The Batch Normalization layer usually comes after a Conv2D layer and normalizes activations within a data batch during training. This ensures consistent input distributions for the next layer, enhancing network learning. It also addresses issues like vanishing or exploding gradients and prevents problems such as feature vanishing or excessive dominance.

During training, it learns the statistics of the input data and saves them as gamma, beta, moving mean, and moving variance. During inference, it assures the input stays inside the statistics it learned by applying equation 2.2, where i is the input of the previous layer, m the moving mean and v the moving variance.

$$\text{output} = \gamma \cdot \left(\frac{i - m}{\sqrt{v + \epsilon}} \right) + \beta \tag{2.2}$$

Note that during inference, if the previous layer is a Conv2D, we can improve inference speed by removing the Batch Normalization layer and merging the weights of both layers. We accomplish this using equations 2.3 and 2.4, where k and b are the original kernel and bias, replaced by the newly merged *Kmerged* and *Bmerged* counterparts.

$$K_{\text{merged}} = \frac{k}{\sqrt{v + \epsilon}} \cdot \gamma \tag{2.3}$$

$$B_{\text{merged}} = \left(\frac{b - m}{\sqrt{v + \epsilon}} \cdot \gamma \right) + \beta \tag{2.4}$$

2.2.3 Max Pooling Layer

The Max Pooling layer (MaxPool2D) is a type of pooling layer that utilizes a sliding evaluation window. It selects the highest value within the window as the output. This down-sampling process reduces the spatial dimensions of the data, condensing information while retaining crucial features.

The output FM size can be calculated using the equation 2.5 [17] where h is height, w width, i input, p pooling window size, and s stride.

$$\text{output FM size} = \left(\lfloor \frac{i_h - p_h}{s_h} \rfloor + 1 \right) \cdot \left(\lfloor \frac{i_w - p_w}{s_w} \rfloor + 1 \right) \tag{2.5}$$

2.2.4 Dense Layer

The Dense layer is fully connected and functions as the decision-making core of a neural network. Each neuron in this layer receives input from all neurons in the previous layer, allowing it to learn relationships between features, [9]. By computing weighted sums of inputs and applying activation functions, the dense layer transforms input data into predictions.

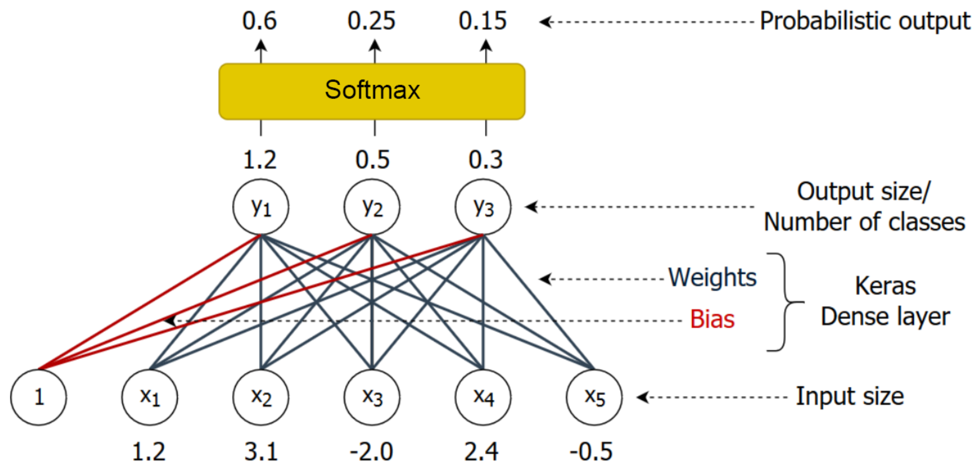


Figure 2.4: Example of dense layer followed by softmax activation. [24]

The output of this layer can be described by the equation 2.6, [8].

$$\text{output} = \text{activation}\,(\text{input} \cdot \text{kernel} + \text{bias}) \tag{2.6}$$

Figure 2.4, illustrates a simple dense layer. At the bottom is the input from the previous layer. Above that, lines in grey and red are the calculations made using the equation 2.6. Ending at the top applying the activation function, in this case, *softmax* that converts arbitrary real values into a probability distribution with values between 0 and 1.

2.3 Recurrent Neural Network

A RNN processes data sequences with sequential information. Tasks like speech recognition, text generation, and video analysis rely on RNNs to consider the previous context, known as the state, to improve prediction accuracy in subsequent steps.

Figure 2.5 illustrates the basic configuration of an RNN. It takes sequential input data, processes it, and produces an output. This output, known as the state, is then fed back into the RNN for the next iteration.

The left side shows a simplified visualization of an RNN, while the right side displays an unrolled version of the same RNN. An RNN can be seen as a series of neural networks trained sequentially using backpropagation, [22].

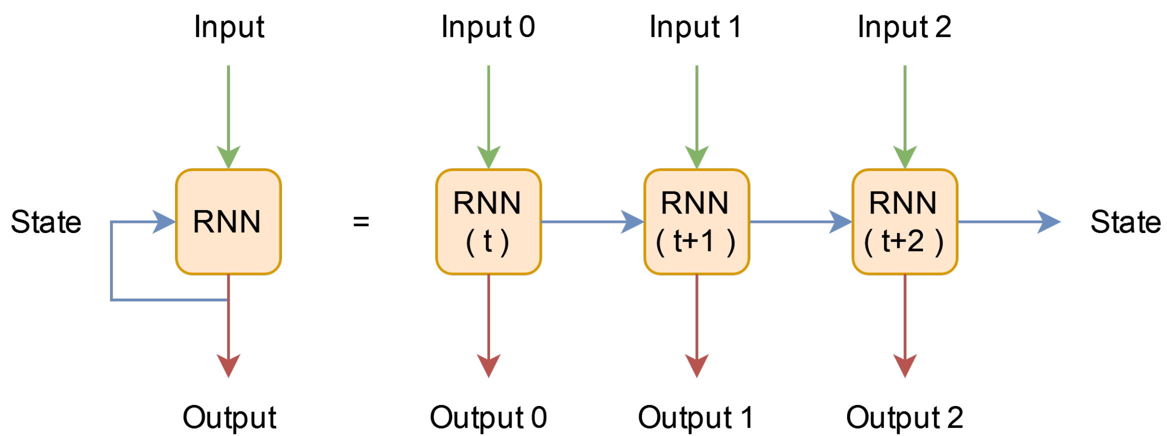


Figure 2.5: Overview of a simple RNN

2.3.1 Gated Recurrent Unit Network

A Gated Recurrent Unit (GRU) is a type of RNN that aims to overcome limitations like the vanishing gradient problem seen in standard RNNs, [14]. This problem occurs when gradients become too small during training, preventing learning [13].

To address this issue, Long Short-Term Memory (LSTM), another type of RNN, introduced gates like the input, forget, cell state update, and output gates. These gates regulate information flow, for updating and discarding said information. GRUs are simpler than LSTMs, featuring just an update gate and a reset gate.

Figure 2.6 shows a GRU cell.

Inputs come from the upper left corner, the previous state enters the bottom, and the output and next state are in the upper right corner. The reset gate, controlled by a sigmoid function, determines how much of the previous state to forget. The update gate, also regulated by a sigmoid function, manages the flow of new information, adjusting the importance of the current input, both in red. Lastly, the tanh activation function, in blue, regulates values within the network, ensuring they remain between -1 and 1. This helps maintain stable gradients during training and avoids issues like the vanishing gradient problem caused by excessively large or small values.

2.3.2 Bidirectional Layer

A RNN relies upon past and present events, but sometimes future events also matter for the overall context of the situation. Take the following sentence:

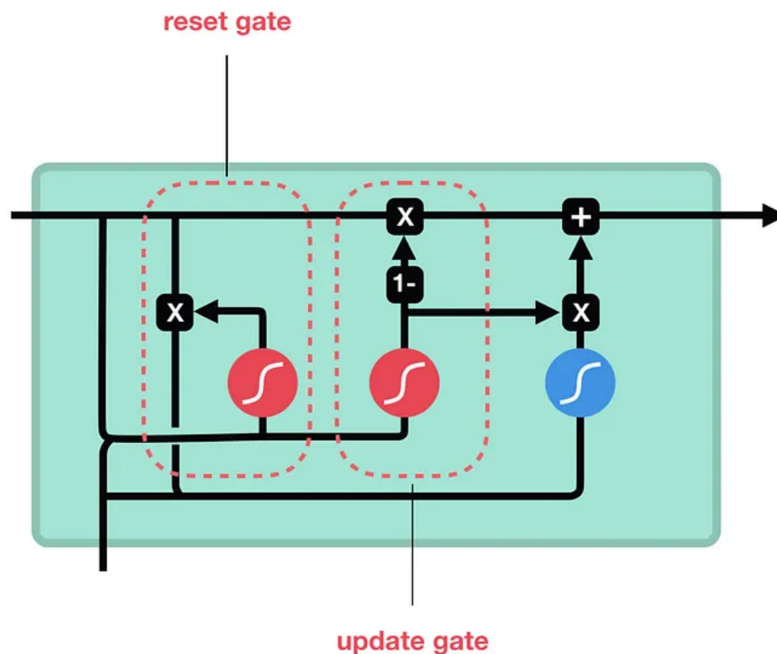


Figure 2.6: GRU overview [15].

"Although John was initially shy, he quickly became the life of the party after he started telling his hilarious stories."

The sentence starts by telling John was shy but by the end of it, John transformed into the life of the party. The future context in this sentence matters to understand the overall idea it wants to present. To provide future context to a RNN, the Bidirectional layer wrapper comes into play, consisting of two separate hidden layers, such as LSTM or GRU layers, with one in the forward direction and another in the backward direction. The forward direction feeds the RNN with input data from start to finish, while the backward direction does it in the opposite direction, from end to beginning.

Using the example provided earlier, for simplicity our RNN accepts 1 word at a time. The forward direction with feed on the RNNs using the following order:

"Although", "John", "was", ...

While the backward direction would feed the RNN:

"stories", "hilarious", "his", ...

Figure 2.7, shows an overview of the architecture. The top row of RNNs is the forward direction, and the bottom is the backward direction.

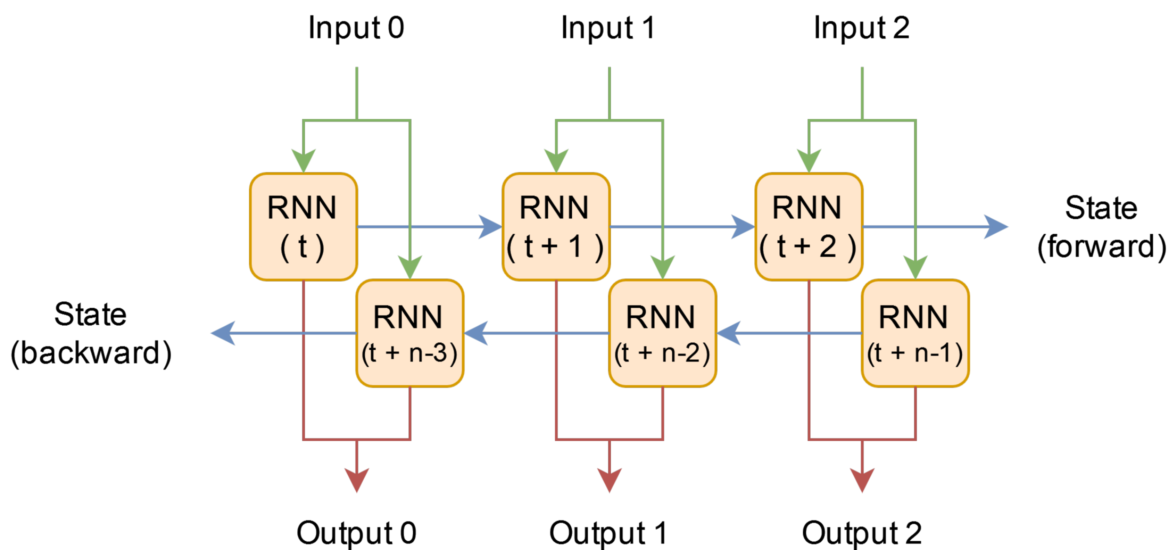


Figure 2.7: Simple bidirectional architecture overview.

2.3.3 Time Distributed Layer

The Time Distributed layer is a wrapper that allows the application of a specific layer. For example, the Dense layer, explained in section 2.2.4, is applied individually to each time step within an input sequence, [25].

Figure 2.8, demonstrates the Time Distributed layer in action using a Dense layer.

The previous layer was a RNN returning three time steps, each with five features. When using the Time-Distributed layer, the Dense layer only sees one time step sequence at a given time. In this example, the Dense layer is set up with only one neuron, returning only 1 output. It is evident, that the output of each time step stands independent from the others, a fact substantiated by the color-coded array displayed on the right-hand side of the figure.

2.4 Model Optimization

To design deep neural models in an embedded system with limited resources, it is important to optimize the neural model to make it more hardware-friendly. One of the most used optimization methods is data quantization.

Quantization is the process of mapping floating-point values to a smaller set of discrete finite values [21]. It is often used in various fields such as signal processing, data compression, and machine learning. In machine learning, quantization can be applied to

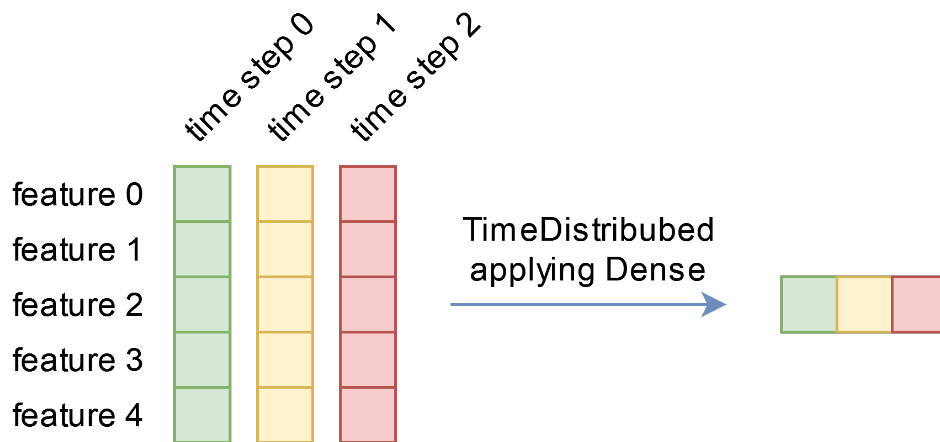


Figure 2.8: Time Distributed using Dense layer overview.

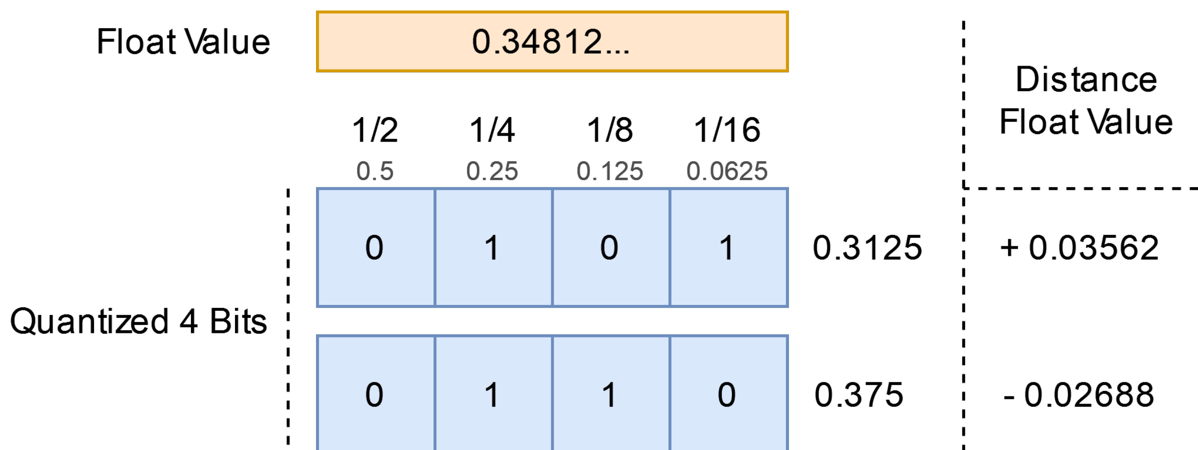


Figure 2.9: Quantization example where floating-point values are mapped to 4 bit fixed-point values

model parameters and activations, to reduce memory usage and arithmetic complexity, while preserving accuracy to some extent.

Figure 2.9, shows an example of a float value being quantized to 4 bits with 0 integers and no sign bit.

Since 0.34812 can not be defined without losing precision, the two closest options are 0.3125 and 0.375. The selected option is 0.3125, which is the closest to the real value.

There are two ways to use quantization in a machine learning model, Quantization-Aware-Training and Post-Training-Quantization.

Quantization-Aware-Training means that the model is trained while considering the quantization process. It learns the quantization-induced error during training, reducing the evaluation error.

In Post-Training-Quantization, the model is trained using the entire range of values expected. After training, the resulting model parameters are then quantized. This process has a higher error margin because the model does not know that the quantization happened, reducing model accuracy.

Both processes reduce model accuracy while improving memory usage and optimizing arithmetic design. QAT is the preferred quantization process because it is the one that introduces the least amount of error during evaluation. On the other hand, PTQ is an option for situations where retraining the model or quantizing some layers is impossible.

2.5 Model Design in FPGA

The current state of deep learning hardware acceleration is vast [16]. However, it is hard to find FPGA projects that tackle the *Bird Audio Detection Challenge*.

The closest project found that did something similar, was a project developed in 2018 by some students from *Rutgers, The State University of New Jersey*. Sadly, the only document found was a poster [23] about the project. This project consisted of the recording *Nocturnal Flight Calls (NFCs)* of migratory birds, creating a CNN that identified the bird's species based on the spectrograms of their Nocturnal Flight Calls, and then organizing the data gathered in a database. It was developed using a *Zybo Z7-10* running *PetaLinux* from *Xilinx*, a custom Linux Distro for this specific hardware made by AMD. The accuracy reported was 70.58%, but was stated that a larger dataset could improve this result.

The project FPGA implementation of an LSTM Neural Network [26] also inspired this work, conducted by José Fonseca from *Faculdade de Engenharia, U.Porto*. Even though is not directly correlated with the work of this thesis, is one that explores a hardware implementation of a Long-Short Term Memory network, with on-chip learning, capability, and flexibility of existing solutions. This project targeted a *XC7Z020, Zynq 7020*, achieving a very good performance compared to the custom-built software implementation by x251 and the current hardware implementation by 14×.

Lastly, the work in [11] aimed to propose a re-configurable framework to implement CNN and RNN models on FPGAs. They tested with 7 different models, achieving an average speedup of 1.3 with a max of 2.83. They used a separate IP core for the CNN and another for the RNN, similar to what was explored in this thesis.

2.6 Conclusion

This chapter introduced the Bird Audio problem and the datasets related to bird detection WHile many implementations based on FPGA exist for these types of models, only a very few works consider the implementation of the bird audio detection problem in FPGA.

In the following chapter, the model considered for BAD is described.

3

Model Design and Optimization

This chapter presents the project workflow from selecting a model to FPGA implementation, how the microfaune_ai model was quantized using QKeras, and the extraction and storage of its weights.

3.1 Workflow from BADC to FPGA

This section outlines the project workflow from the moment of selecting a model, quantizing it, and ending with a FPGA implementation. This workflow is shown in figure 3.1.

The workflow starts by finding bird audio detection models that are related to the BADC. Then find out which ones are suitable for hardware implementation and validate their accuracy against the reported by the model developer.

With the model selected, its weights are extracted using a Python script. At this stage, these weights are still in floating-point format, but they serve to validate the replicated model in C. Additionally, the Python model is executed to evaluate some input audio files, allowing the extraction of the output of each layer to further validate the C implementation.

The next step is to quantize the model to reduce the memory footprint used by the weights and the complexity of the arithmetic operators while trying not to reduce the original model accuracy. The quantization is done with QKeras considering a

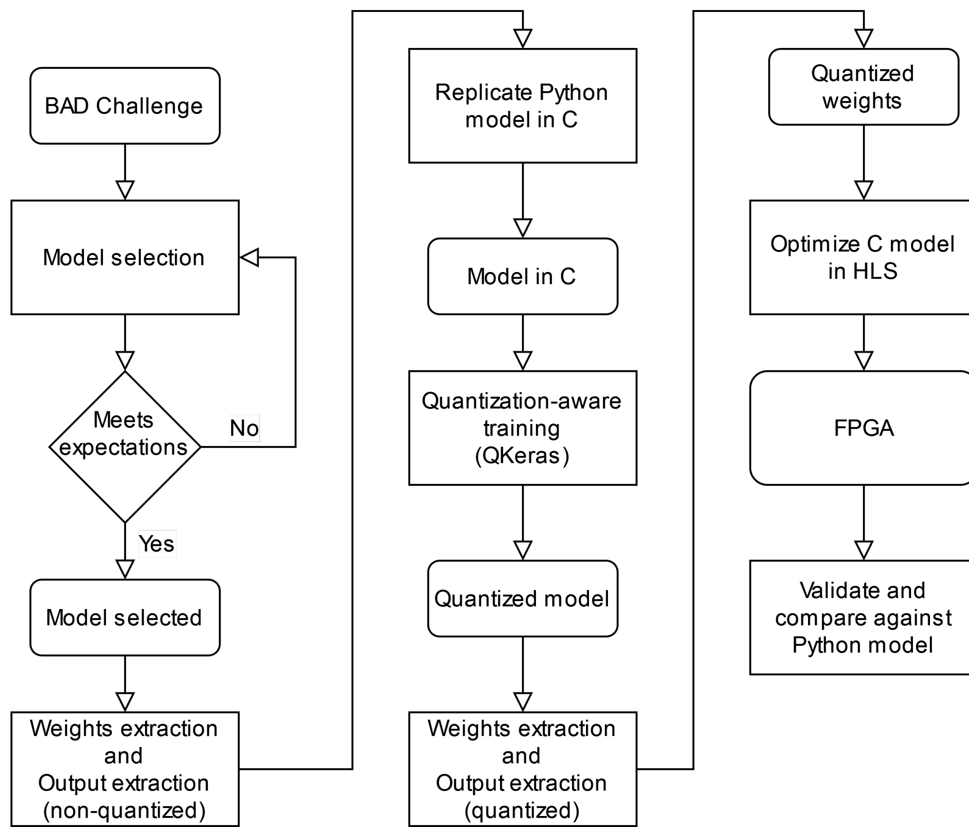


Figure 3.1: Project workflow

Quantization-Aware Training approach. Multiple quantizations with different data bitwidths are explored to determine the minimum amount of bits required for network operation with minimal accuracy reduction.

The quantized weights are then extracted using the weights extraction script, now adapted to support QKeras quantized weights. The output is also extracted for each layer, with the same purpose as before, to validate the next step of replicating the Python model in C but this time with quantized weights and activations.

Optimizing the model in High-Level Synthesis (HLS) involves validating the quantized model against the extracted output, and then applying the HLS optimizations such as pipelining, cycle unrolling, and array partitions.

For FPGA deployment, one or more Intellectual Property (IP)s are designed with the HLS model code and then programmed into the FPGA.

Finally, a comparison between the FPGA-developed algorithm and the original Python variant is made, assessing both accuracy and speed.

3.2 Model Selection

This section explores some models that entered the BADC, the challenges and successes of the execution of said models, and the selection of the model to be implemented.

From the many deep learning models found targeting the bird audio detection problem with available code, the following were considered the most relevant:

- BADC-2017[1].
- BAD2[2]
- bird_audio_detection_challenge[3]
- bird-audio-detection[4]
- All-Conv-Net-for-Bird-Activity-Detection-Significance-of-Learned-Pooling[5]
- microfaune_ai[6]

From this set of models, only the last two were successfully replicated.

3.2.1 Model: All-Conv-Net-for-Bird-Activity-Detection-Significance-of-Learned-Pooling

ThisAll-Conv-Net-for-Bird-Activity-Detection-Significance-of-Learned-Pooling (ACNet) model came with two variants:

1. **Max Pool Variant (MPV)** - This model aggregates the information present in each feature-map individually [1]. To accomplish this they use Max Pooling layers;
2. **Learned Pool Variant (LPV)** - This model takes into account the inter feature-map correlations which are ignored in traditional max-pooling" [1]. This variant uses Convolution layers in place of Max Pooling.

Figures 3.2 and 3.3 show both model variants.

[1]BADC-2017: https://github.com/karolpiczak/BADC-2017
[2]BAD2: https://github.com/himaivan/BAD2
[3]bird_audio_detection_challenge: https://github.com/topel/bird_audio_detection_challenge
[4]bird-audio-detection: https://github.com/mvrl/bird-audio-detection
[5]ACNet: https://github.com/arjunp17/All-Conv-Net-for-Bird-Activity-Detection-Significance-of-Learned-Pooling
[6]microfaune_ai: https://github.com/microfaune/microfaune_ai

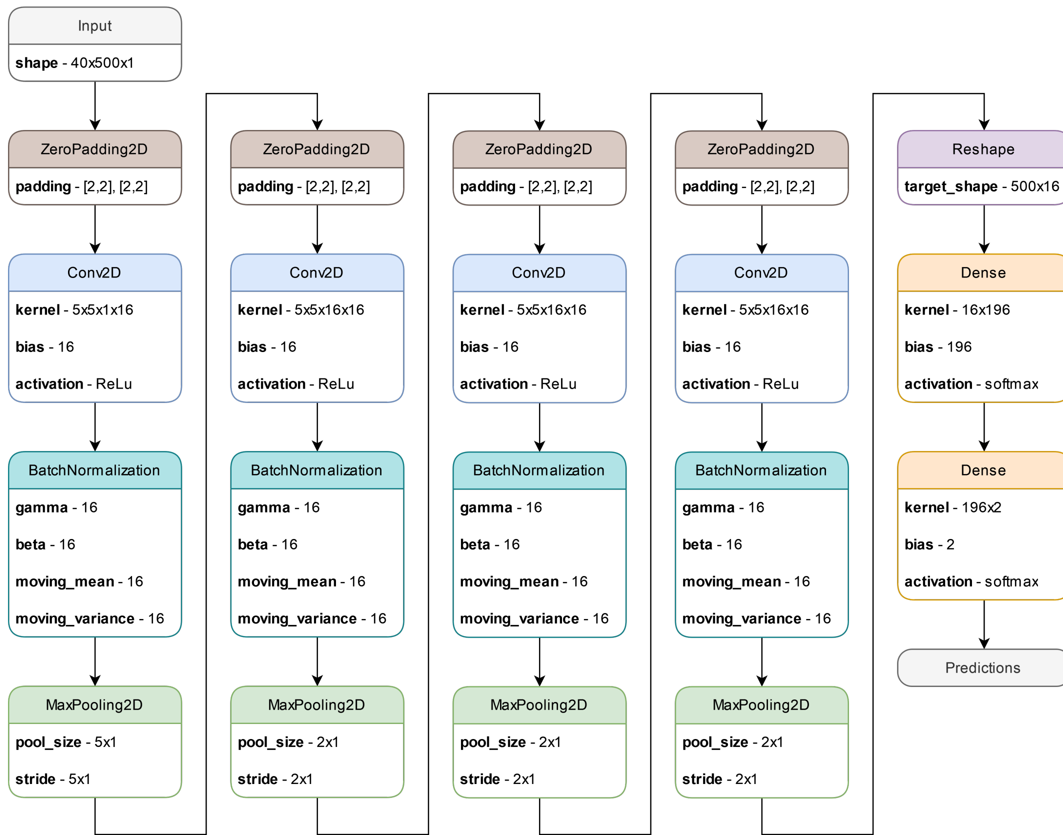


Figure 3.2: Model: ACNet MPV

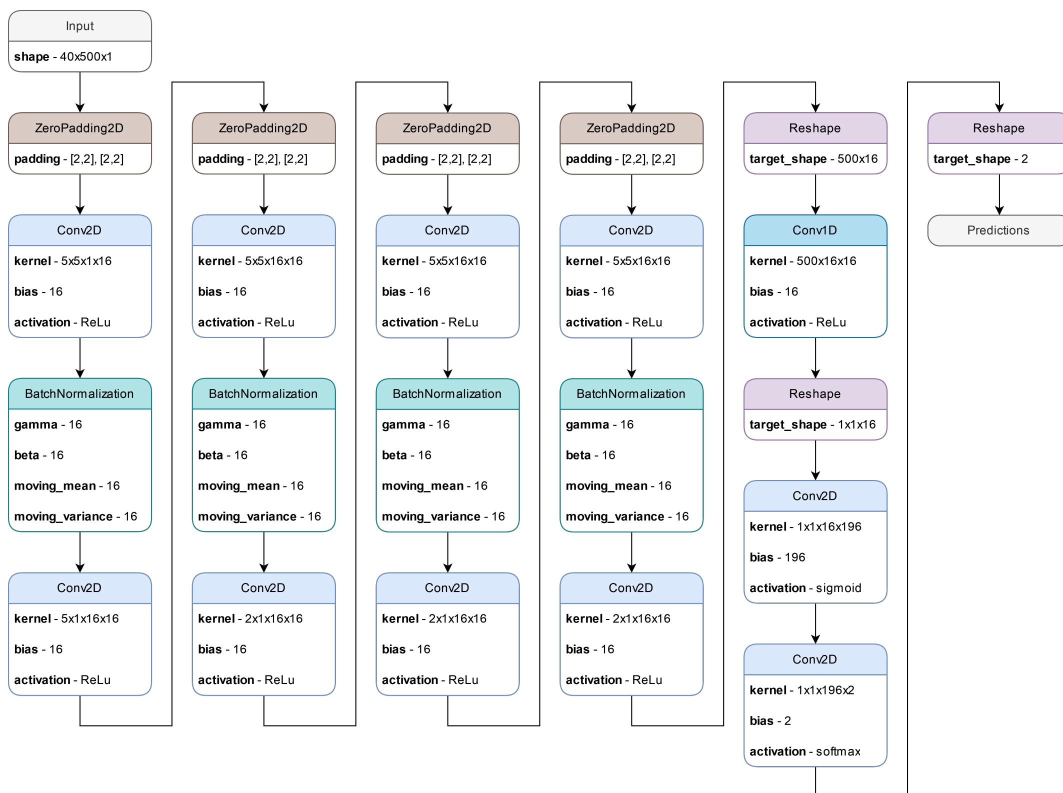


Figure 3.3: Model: ACNet LPV

These two model variants required training and fine-tuning of parameters `test_size` and `epochs`. This adjustment was prompted by the realization that the original code used an invalid value for `test_size`, while TensorFlow mandates values within the range of 0.0 to 1.0. Throughout this process, the `batch_size` remained unaltered to its original value of 32.

Code 3.1 shows the model settings used for training.

```
#train_data
...
x_tr, x_tst, y_tr, y_tst = train_test_split(feature, label, test_size=0.2,
    shuffle=True)
...
#fit the model
hist = model.fit(x_tr, y_tr, epochs=50, batch_size=32, verbose=2)
```

Listing 3.1: Model setting used for training.

Code 3.2 shows the changes made.

```
# before all the imports at the start of the file
import os
os.environ['PYTHONHASHSEED']=str(seed)
os.environ['TF_DETERMINISTIC_OPS'] = '1'
...
# after all the imports
tf.compat.v1.reset_default_graph()
tf.random.set_seed(seed)
tf.keras.utils.set_random_seed(seed)
np.random.seed(seed)
random.seed(seed)
```

Listing 3.2: Changes to the original code adding support for setting a seed.

The next step was to find the best seed to verify their claims on the MPV vs LPV, and identify the best candidate model to be implemented in hardware. To accomplish this, a script was developed, and the following parameters were manipulated:

- **`seed`** - values in the range [0..50[, for no particular reason, other than the fact of balancing overall time and number of runs.
- **`test_size`** - set to 0.2, 20% train and 80% test.
- **`epochs`** - started with 50, the original value set by the developer of the model, then went to 100, ending with 200. To check if a higher number of `epochs` would impact positively or negatively the trained model.

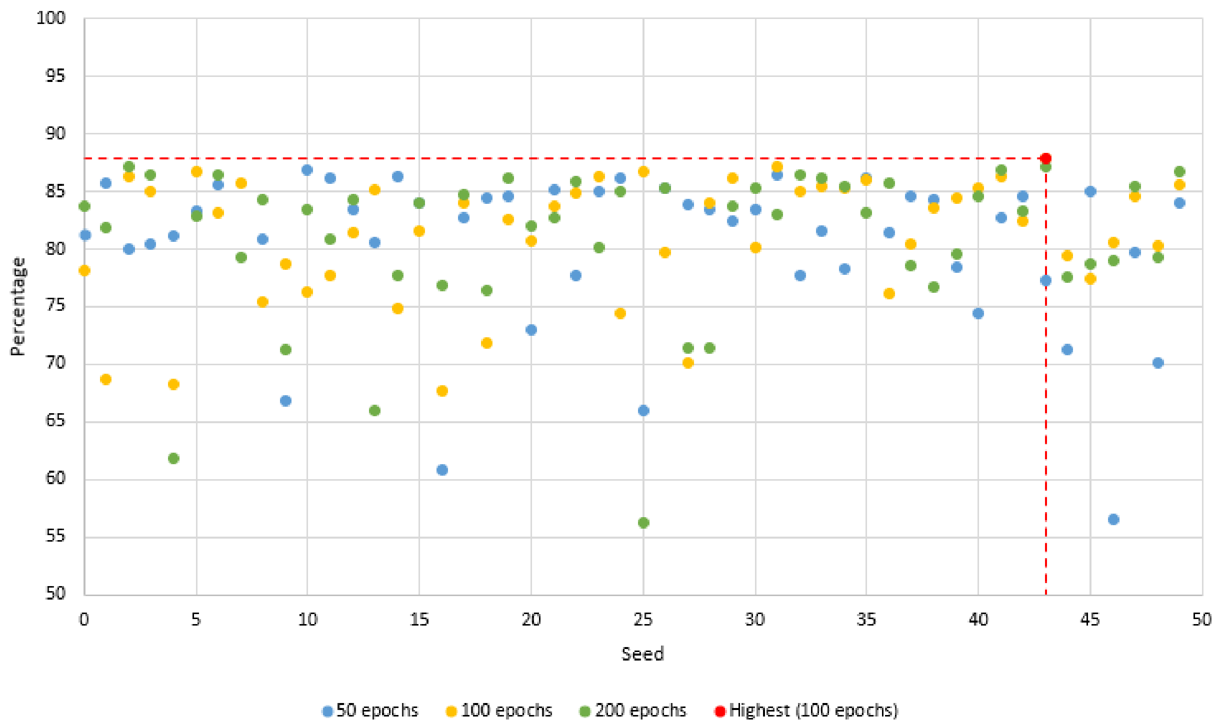


Figure 3.4: Seed search results for MPV, 87.9%.

The optimal seed found for the MPV was `43`, with `epochs` set to `100`, resulting in an accuracy of 87.9%, shown in figure 3.4 with a red dot.

For the LPV, the best seed found was `35`, with `epochs` set to `50`, giving an accuracy of 86.1%, shown in figure 3.5 with a red dot.

The developer's paper [1] presents an expected accuracy difference between LPV and MPV, 88.91% and 85.01% respectively. In our design, the LPV achieved 86.1% accuracy and MPV 87.9%.

Comparing the amount of weights, the MPV uses 23390 weights while the LPV uses 154286, 6.59 times more resources than the MPV.

Given the accuracy and the number of required weights, the candidate variant for hardware implementation would be the MPV. It had a higher accuracy, and when comparing the model structure and amount of weights, the MPV is simpler and uses fewer weights, leading to less memory usage.

3.2.2 Model: microfaune_ai

The microfaune_ai [19] model is based on the microfaune [18], with a reported accuracy of 90.18%. It did not enter the BADC but uses the same datasets for its training.

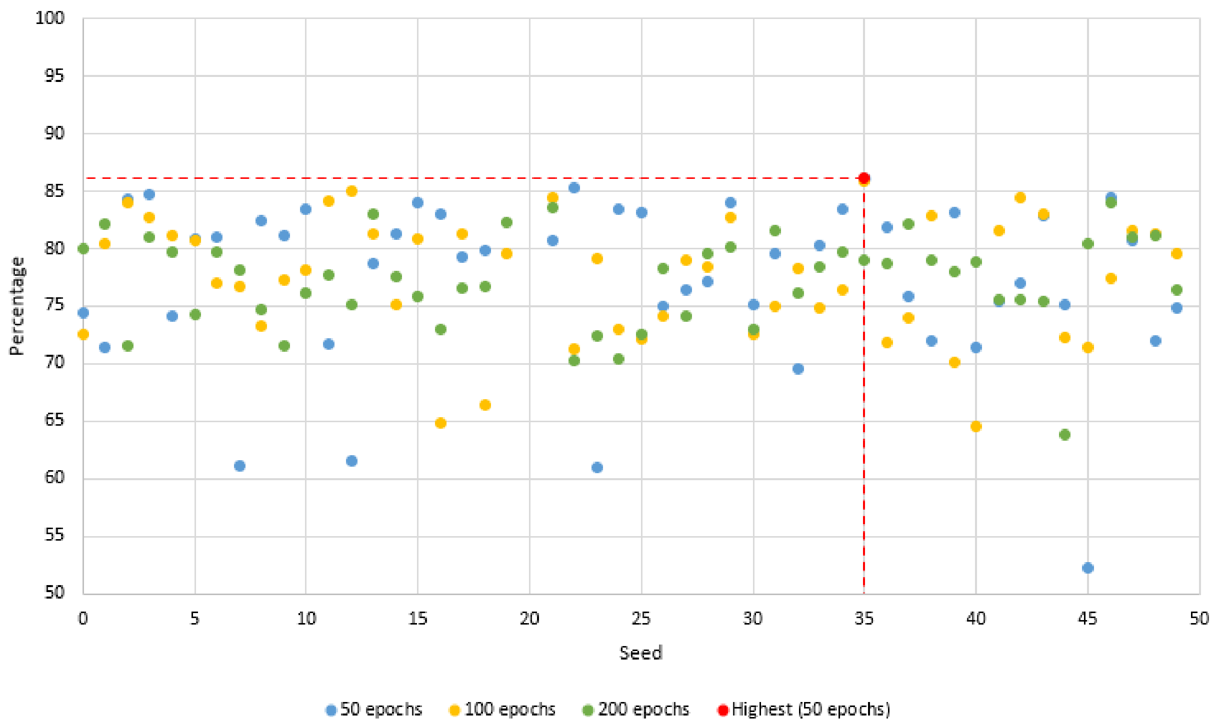


Figure 3.5: Seed search results for LPV, 86.1%.

The model is composed of a CNN section with Convolutions and Max Pooling, a RNN section with Bidirectional GRUs, and ends with Time-Distributed Dense layers.

Figure 3.6 shows the model diagram.

This model implementation is available through Python's package manager and accepts WAV audio files:

```
pip install microfaune-ai
```

Similarly to the ACNet model, this one also lacked a seed. Following the model training guide `learn_model.ipynb` found in the microfaune repository, was added a seed and a variable number of epochs for training as follows:

- **`seed`** - values in the range [0..10[. Initially, the range was set to [0..50[, but after some iterations, it became evident that the model's accuracy followed a pattern based on the seed, shown in figure 3.7. This reduced the seed search time.
- **`epochs`** - the original value was 100, but 50 epochs were also checked because the model remained stable after around the 40th epoch.

Figure 3.7 shows the seed accuracy pattern and the best seed marked in red.

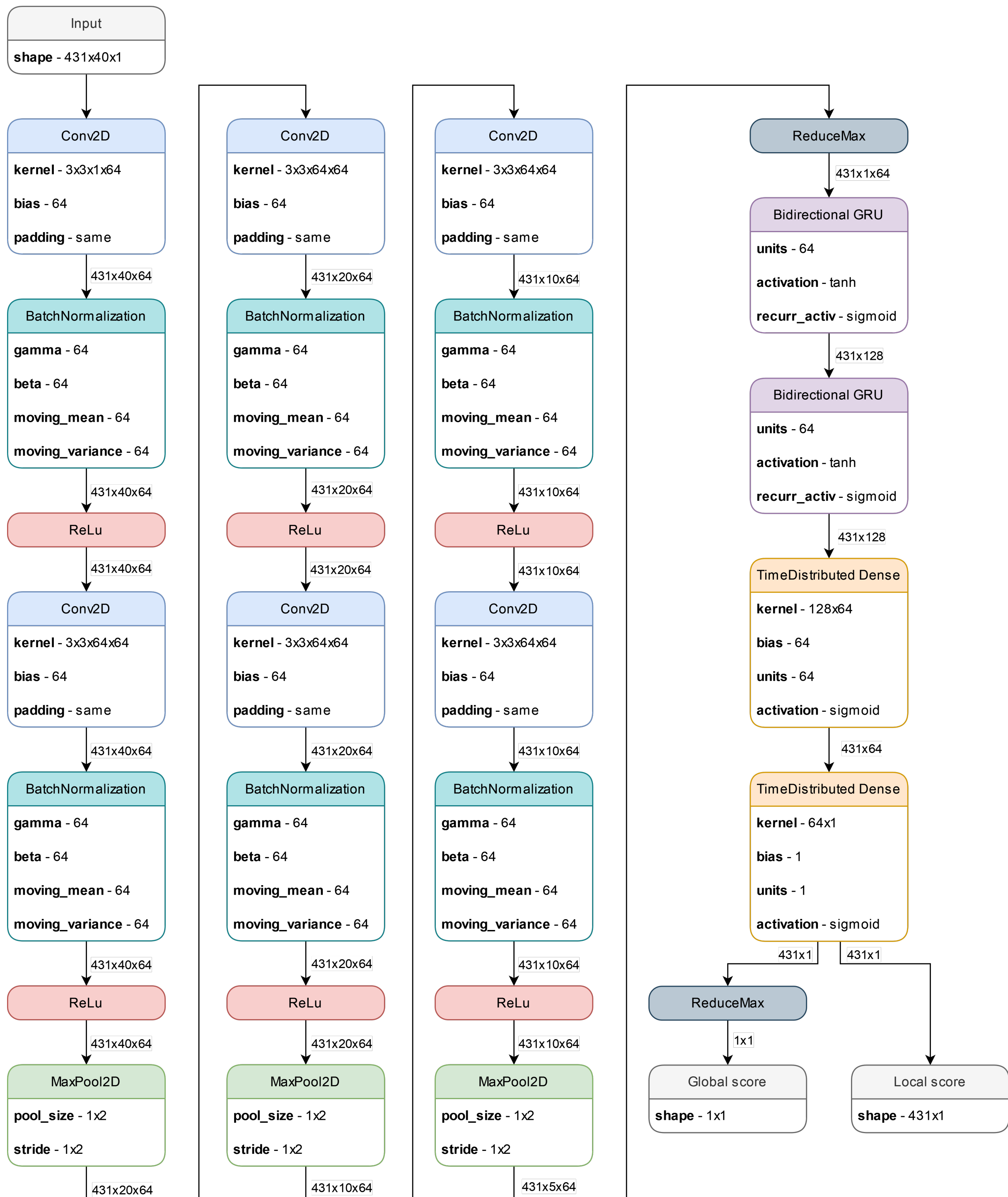


Figure 3.6: Model: microfaune_ai

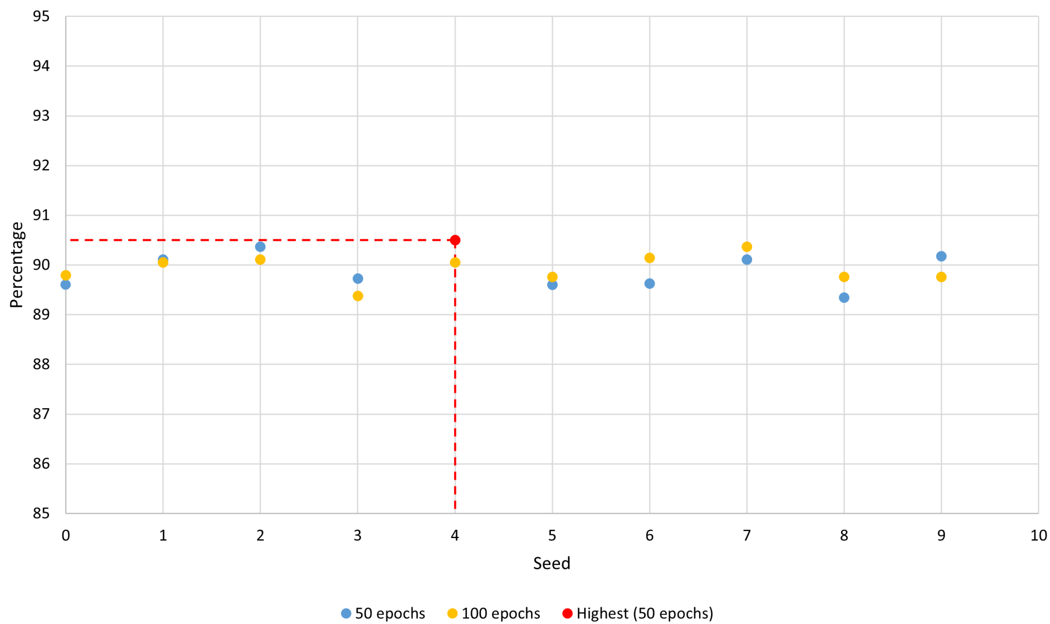


Figure 3.7: Seed search results for *microfaune*

The optimal seed found was `4`, with `epochs` set to `50`, resulting in an accuracy of 90.5%, shown in figure 3.7 with a red dot. The seed pattern can also be seen.

The amount of weights of the model is 316481.

The model's accuracy was originally documented as 90.18%. However, through the `seed` search process, it was discovered that using `seed` 4 resulted in an accuracy of 90.5%. This outcome closely corresponds to the expected accuracy level.

The ACNet model variants MPV and LPV have an accuracy of 87.9% and 86.1% respectively. The microfaune_ai model had an accuracy of 90.5%.

Microfaune_ai has 316481 weights, while the ACNet MPV uses 23390 weights, and the LPV uses 154286. Even though it uses more weights, and thus more memory, it was the selected model. It had the best accuracy and the weights can be all stored on-chip memory even in a small density FPGA.

3.3 Exploring Quantization Techniques

Two approaches were followed to quantize the model. The first one uses the TensorFlow model optimization that converts a TensorFlow model into a TFLite[7] model. This TFLite model is aimed at optimizing the models to run on small devices, like mobile

[7]TensorFlow Lite model: https://www.tensorflow.org/lite/models/convert

phones, that run a small version of TensorFlow intended only for inference. This optimized[8] model uses less processing time and RAM. However, this approach as limited control over the number of bits for model quantization considering pre-defined types: Int8, Int16, Float16, and other byte-aligned types. Another limitation was not being able to quantize the model partially.

The second approach is based on the QKeras[9] library, a quantization extension to Keras that provides a drop-in replacement for some of the Keras layers to quickly create a deep quantized version of Keras network [20].

The QKeras library allows individual selection of layers for quantization and the specification of the number of bits for the quantization process. The library requires some improvements when saving the model weights. The function[10] saves the quantized weights to a TensorFlow format, `H5`, not useful for hardware implementation. This function also lacks the possibility of merging the weights with their scale. These limitations were overcome during this project by developing custom Python scripts to save the weights in a useful format to be implemented in hardware.

3.4 Training Microfaune_ai with QKeras

Different quantization configurations were tested with QKeras by varying the number of bits across the multiple layers of the model to find the best-balanced option between model size reduction and accuracy.

The original microfaune_ai model is in TensorFlow and uses Keras layers that had to be replaced by QKeras layers, namely:

- **Conv2D** and **BatchNormalization** replaced by **QConv2DBatchnorm**
- **ReLU** replaced by **QActivation** configured as ReLU

The Bidirectional GRU layers could be replaced by QBidirectional and QGRU respectively. However, during testing, it was observed that the QKeras implementation of the GRU layer is based on GRUv1 rather than the GRUv2 used by the microfaune_ai model. The Dense layer could be replaced by the QDense layer, but it is used inside the

[8] TensorFlow model: https://www.tensorflow.org/lite/performance/model_optimization

[9] QKeras GitHub repository: https://github.com/google/qkeras

[10] QKeras model_save_quantized_weights: https://github.com/google/qkeras/blob/6be2f5ca75d9a42209b17f42e86087f6d60cf961/qkeras/utils.py#L222

Time-Distributed layer which is not implemented in QKeras. MaxPool2D and Reduce-Max layers did not require layer replacement, since they work and select the output values from the previous layers, already quantized.

An additional layer was added before the first convolutional layer to quantize the input image. This does not affect the original value other than forcing a quantization, because the input values range from 0 to 1. Listing 3.3 shows the added QActivation, line 2, after the input setup.

```
spec = keras.Input(shape=[431, 40, 1], dtype=np.float32)    # no changes
x = QActivation(f"quantized_relu(4,0)")(spec)
```

Listing 3.3: Added ReLu activation to quantize the input of the first Conv2D.

To quantize the CNN section of the model, the Conv2D and BatchNormalization layers were replaced by QConv2DBatchnorm with symmetric quantization. The symmetric option set to true means that the range of values at the positive range is equal to the negative side. Output values range from -1 to 1. Listing 3.4 shows the replacement layer and its configuration.

```
#### Original microfaune_ai code ####
conv_reg = keras.regularizers.l2(1e-3)
# x -> previous layer
x = keras.layers.Conv2D(64, (3, 3), padding="same",
    kernel_regularizer=conv_reg, activation=None
)(spec)
x = keras.layers.BatchNormalization(momentum=0.95)(x)

#### QKeras replacement code ####
# x -> previous layer
x = QConv2DBatchnorm(64, (3, 3), padding="same",
    kernel_regularizer=conv_reg, activation=None,
    kernel_quantizer = "quantized_bits(4,1,1)",
    bias_quantizer = "quantized_bits(4,1,1)",
    momentum=0.95
)(x)
```

Listing 3.4: Conv2D and BatchNormalization replaced by QConv2DBatchnorm.

The ReLU activation after each BatchNormalization layer was replaced by QActivation configured as ReLU. Output values range from 0 to 1. Listing 3.5 shows the replacement layer and its configuration.

```
#### Original microfaune_ai code ####
x = layers.ReLU()(x)

```

```
#### QKeras replacement code ####
# x -> previous layer
x = QActivation(f"quantized_relu(4,0)")(x)
```

Listing 3.5: ReLu activation replaced by QActivation configured as ReLu.

The Bidirectional GRU was not quantized during training because QKeras GRU implementation is based on the GRUv1 of Keras and not the GRUv2 used by microfaune_ai. For this reason, these layers were quantized after model training by truncation to 8 bits with 1 integer, Q(8,1). However, if a future version of QKeras implements GRUv2, the proposed replacement code can be seen in listing 3.6.

```
#### Original microfaune_ai code ####
x = layers.Bidirectional(layers.GRU(64, return_sequences=True))(x)

#### QKeras replacement code ####
# x -> previous layer
x = QBidirectional(
    QGRU(units = 1,
         activation = "quantized_tanh(8)",
         recurrent_activation = "hard_sigmoid(8)",
         kernel_quantizer = "quantized_bits(8,1,1)",
         recurrent_quantizer = "quantized_bits(8,1,1)",
         bias_quantizer = "quantized_bits(8,1,1)",
         state_quantizer = "quantized_bits(8,1,1)",
         return_sequences = True
    )
)(x)
```

Listing 3.6: Possible Bidirectional GRU replacement with QBidirectional and QGRU.

The Time-Distributed Dense was not quantized because QKeras currently does not implement a quantization layer for Time-Distributed. The Dense layer can be quantized with QDense, but since it is inside a Time-Distributed, QKeras cannot quantize it. It was decided that these layers were not going to be quantized and executed on the ARM processor using floats instead. This decision comes from the small layer size of both TD Dense, at the end of the model, and to reduce further impact on the model accuracy.

The QKeras model was trained in different quantization settings with a total of 100 epochs and 100 steps per epoch. As the tests progressed, the reduction of the number of bits affects the initial convergence of the training (See figures 3.8, 3.11, 3.9, 3.10).

Figure 3.8: Original microfaune_ai, accuracy training progression plot, 90.50%.

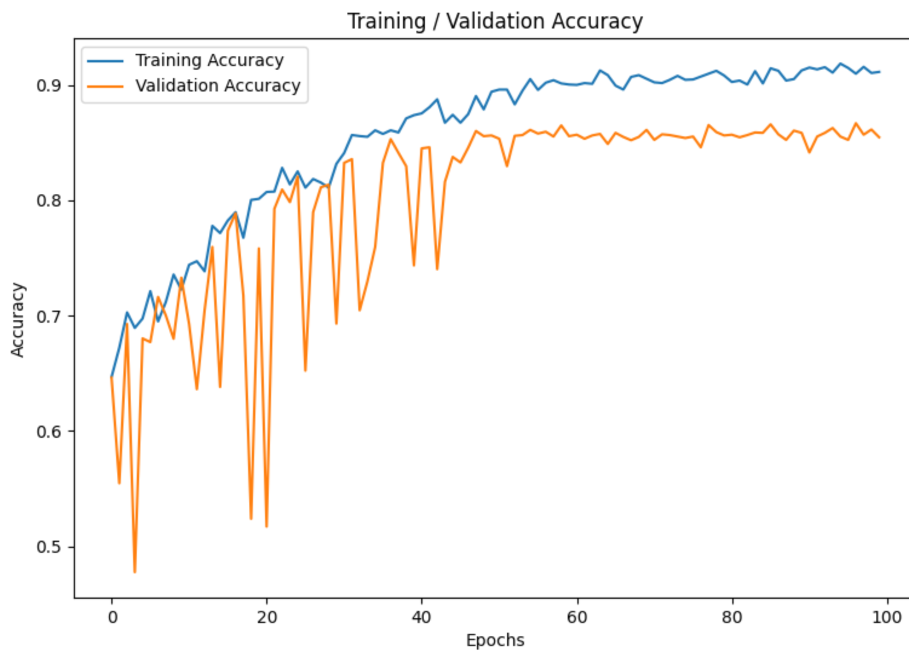


Figure 3.9: CNN quantized to 4 bits and RNN 64 cells, accuracy training progression plot, 85.46%.

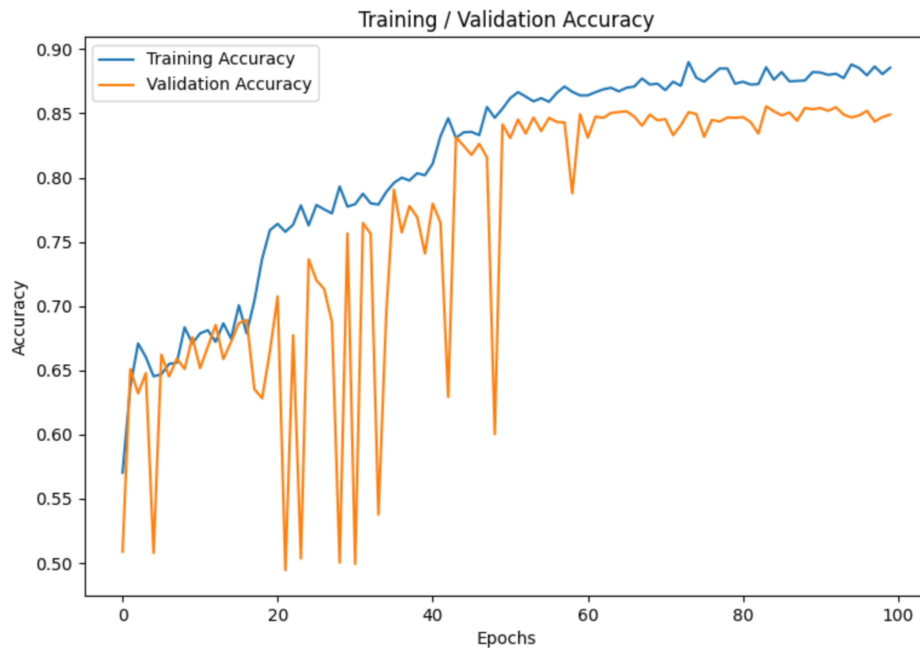


Figure 3.10: CNN quantized to 4 bits and RNN 1 cell, accuracy training progression plot, 84.91%.

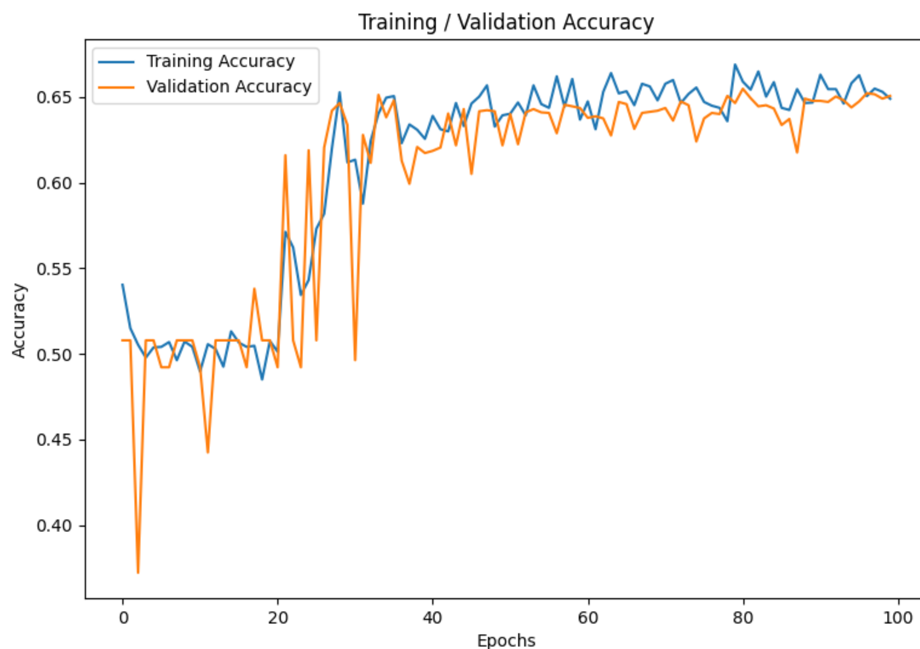


Figure 3.11: CNN quantized to 2 bits and RNN 64 cells, accuracy training progression plot, 65.07%.

	Original	*Quantized*			
Quant.bits ǀ GRU cells	**- ǀ 64**	**16 ǀ 64**	**8 ǀ 64**	**4 ǀ 64**	**2 ǀ 64**
Accuracy	90.50%	88.60%	85.97%	**85.46%**	65.07%

Table 3.1: Accuracy comparison between different amounts of bits used for quantization.

The most interesting results were:

- CNN quantized to 2 bits with 1 integer. RNN to float with 64 GRU cells. Fig 3.11.
- CNN quantized to 4 bits with 1 integer. RNN to float with 64 GRU cells. Fig 3.9.
- CNN quantized to 4 bits with 1 integer. RNN to float with 1 GRU cell. Fig 3.10.

The lowest number of bits achieved for quantization of the CNN section of the model was 4 bits with 1 integer. The difference in accuracy between using 16 bits and 4 bits is around 3.5%, and when comparing 8 bits to 4 bits, around 1% less accurate. When using 2 bits with 1 integer for CNN quantization, the model accuracy dropped drastically. The accuracy of the model quantized with 4-bit compared to the original model dropped from 90.50% to 85.46%, a 5% reduction in accuracy.

Table 3.1 shows the multiple training done at various bit configurations.

Considering that the accuracy between using a quantization of 4 and of 8 bits is less than 1%, the 4 bit quantization was chosen because it requires around three times less computing resources for the same performance.

The RNN section of the model was also optimized through model size reduction and post-training quantization. To reduce the model size, the number of GRU cells was reduced from 64 to a number ranging from 1 to 64 in powers of 2. From the training results gathered, it was concluded that the number of GRU cells marginally affects the model accuracy (See Table 3.2).

The quantized model accuracy with 64 cells is 85.46%, and with 1 cell is 84.91%, a reduction of only 0.55%. This drastically reduces the number of weights used by the RNN section of the model as shown in table 3.3. The number of Bidirectional GRU parameters was reduced by a factor of 288 when using 1 GRU cell as opposed to 64 cells.

	Original	*Quantized*						
GRU Cells	**64**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Accuracy	90.50%	85.46%	84.75%	85.33%	83.95%	85.23%	83.18%	**84.91%**

Table 3.2: Accuracy comparison between different amounts of GRU cells used, CNN quantized to 4 bits and RNN to 8 bits.

	64 Cells		1 Cell	
Number Params	**BGRU_0**	**BGRU_1**	**BGRU_0**	**BGRU_1**
Forward Bias	192	192	3	3
F. Recurrent Bias	192	192	3	3
F. Kernel	12288	24576	192	6
F. Recurrent Kernel	12288	12288	3	3
Backward Bias	192	192	3	3
B. Recurrent Bias	192	192	3	3
B. Kernel	12288	24576	192	6
B. Recurrent Kernel	12288	12288	3	3
Total		124416		432
	Reduction Factor			288

Table 3.3: Comparison of the number of parameters used by the Bidirectional GRUs with 64 cells vs 1 cell.

3.5 Weights Extraction

QKeras provides function `model_save_quantized_weights`[11] to extract the model weights after quantization. However, this function saves the weights in the TensorFlow format which is not useful for an FPGA implementation. For this reason, based on this function, a Python script was created to extract the model weights and store them sequentially in a binary file.

After loading the model, the `layer` array containing the layer information and the `weight` array containing the quantized weights can be accessed. This `weight` array will give us the layer weights and bias. To extract the scales of the weights, QKeras provides the function `get_quantizers`. A new function `getQuantizeScale` that facilitates the usage of this function was developed.

These quantizers have a `scale` for each quantized set of weights. In the case of QConv2DBatchnorm it will return `kernel_scale` and `bias_scale`. The `bias_scale` is ignored when using the microfaune_ai QKeras model since it only returns a single value of 1, meaning no scaling is required.

[11]QKeras "model_save_quantized_weights": https://github.com/google/qkeras/blob/6be2f5ca75d9a42209b17f42e86087f6d60cf961/qkeras/utils.py#L222

Before saving the weights and scales to a binary file, the kernel and scale are merged to give the scaled kernel value, removing this necessary step from the FPGA implementation that only uses integers calculations. A merge kernel with scale function, `mergeKernelScale` (Listing 3.7), was created that receives the kernel and the scale, divides the kernel array by the scale array, and then multiplies the resulting array by the value 4 at each position, returning an array of integers called `kernelWithScale`.

```
1 def mergeKernelScale(kernel, scale):
2     # kernel and scale are a numpy arrays
3     return (kernel / scale) * 4
```

Listing 3.7: Function mergeKernelScale.

The bias needs to be adjusted to be used along with the kernel merged with scale. It was concluded that in the FPGA implementation 16 bits are enough to store the internal accumulations. A function was created, `createBiasHLS`, to to adjust the bias to the format of the accumulator so that the initialization of the accumulator is a direct copy of the bias to it. The function multiplies the bias array by 4, the same reason as the kernel and scale situation, and then adjusts the bias where it is multiplied by 2 raised to the power of the sum of 2 with scaleHLS. Ends by returning this new bias array called, `biasHLS`.

```
1 def createBiasHLS(bias, scaleHLS):
2     # bias and scaleHLS are numpy arrays
3     return np.power(2, 2+scaleHLS) * (bias*4)
```

Listing 3.8: Function createScaleHLS.

Considering the first layer of the QKeras microfaune model, the QConv2DBatchnorm layer, three binary files were generated, the kernel, the bias, and the kernel_scale. Listing 3.9 demonstrates a simplified weights extraction for the first layer of the microfaune model, q_conv2d_batchnorm_0.

```
1  model    # the loaded model
2
3  layer = model.layers[1]          # 0 the input layer, 1 the 1st layer
4  weights = layer.weights          # array with all weights of layer
5  layerName = layer.name           # layer name, ex: 'q_conv2d_batchnorm_0'
6  in_shape = layer.input.shape     # shape of the input into this layer
7  out_shape = layer.output.shape   # shape of the output out of this layer
8
9  kernel_shape = weights[0].shape     # shape of the kernel
10 kernel = weights[0].numpy()         # kernel values
11
12 bias_shape = weights[1].shape       # shape of the bias
```

```
13  bias = weights[1].numpy()          # bias values
14
15  kScale = getQuantizeScale(layerName)      # get kernel scale values
16  kernelWscale = mergeKernelScale(kernel, kScale) # merge kernel w/ scale
17  scaleHLS = createScaleHLS(kScale)         # scale to FPGA scale
18  biasHLS = createBiasHLS(bias, scaleHLS)   # bias to FPGA bias
19
20  saveToBinary(kernelWscale)       # binary file with merged kernel and bias
21  saveToBinary(scaleHLS)           # binary file with FPGA scale
22  saveToBinary(biasHLS)            # binary file with FPGA bias
```

Listing 3.9: Example of extracting weights from the 1st QConv2DBatchnorm.

In this example, the trained model was loaded into the `model` variable containing the layers information and weights under `layers`. Object `layers` provides an array, the first position being the input layer, and the second the QConv2DBatchnorm layer at position 1. The input and output sizes can be accessed at `layer.input.shape` and `layer.output.shape` respectively. The `weights` array contains the information and data for each type of weight. In the case of the QConv2DBatchnorm layer, there are 4 types: kernel, kernel_scale, bias, and bias_scale. The bias_scale is ignored, as explained above. The retrieved information is then arranged and saved in a file for later use.

In the case of non-quantized layers, the same workflow applies but without getting the scale and merging weights. The kernel and bias are saved directly to a binary file, skipping the lines from 15 to 18 on the listing 3.9 example.

The final step is to save this output into a binary file to validate the C model layers later.

3.6 Conclusion

This chapter presented the project workflow, the model selection, the quantization process, and the extraction of weights. Explained in detail how the microfaune_ai model was quantized using QKeras by replacing some of the Keras layers with QKeras layers, and how its weights were extracted and stored into separate binary files for later use in the FPGA implementation.

Model reduction and quantization optimizations were applied to the microfaune_ai model. The final model after quantization has a single RNN cell in the RNN layers and weights/activations quantized with 4 bits.

4

Embedded Software Microfaune Model

This chapter presents the process of rewriting the model from Python to C. Each layer adaptation is explained alongside its pseudo-code. The GRU layer has higher depth in its explanation since it is the least explored in FPGA related literature.

4.1 TensorFlow Layers Independent Implementation in Python

To implement the microfaune model in C each layer had to be individually replicated and tested. The step-by-step procedure involved:

1. Creating a simple test environment in Python using TensorFlow.
2. Reverse engineering or finding examples of how they work, and implementing them using the Numpy library.
3. Reworking the model code from Python to C and validating with the test environment.

The next stage was using the Numpy library and removing TensorFlow. The output of the replicated layer was expected to match the original and a necessary step before transitioning to the C implementation.

4.1.1 Conv2D

The microfaune_ai model starts with 6 Conv2D layers, with the first one receiving an input of 1 channel outputing 64 channels, and the rest 64 channels for input and output. Reworking the Conv2D layer, there were two options, calculate 1 filter at a time or calculate all the 64 filters simultaneously. The first option was easier to understand, but eventually the second option was chosen. The reason is that the TensorFlow Conv2D layer generates an output ordered by filter and then by line and column. The other reason for selecting the second option was a previously developed generic Conv2D from a colleague from ISEL, only requiring some adjustments to target our specific problem.

The Conv2D used in the model have their number of input columns reduced every two Conv2D, because of the presence of MaxPool2D layers. The number of inputs 431 lines and 64 channels, remains constant until the end of the CNN section of the model.

The general outline of the replicated Conv2D in C can described in algorithm 1.

The algorithm iterates through the lines, and columns and then filters. For simplification, this pseudo-code represents the Conv2D layers after the first Conv2D of the model, using the entire 64 channels as input. Then, the kernel and bias are applied at that line and column for the 64 filters, accumulating the result. The ReLu activation is applied to the accumulated value. The output is then returned following the same order as the input, all the 64 calculations on each filter for a line and a column, repeating this order. The accumulator is reset to 0 after the output is calculated.

Algorithm 1 Conv3D

```
 1: function CONV3D(filters, input, columns, kernels, bias)
 2:     for line ← 1 to (433 − 1) do                          ▷ 433 = 431 + padding
 3:         for column ← 1 to columns do                      ▷ including padding
 4:             for filter ← 0 to (filters − 1) do
 5:                 currBias ← bias[filter]
 6:                 acc ← currBias
 7:                 for kline ← 0 to 3 do                     ▷ kernel line iteration
 8:                     for kcol ← 0 to 3 do                  ▷ kernel column iteration
 9:                         for channel ← 0 to (64 − 1) do    ▷ 64 channels
10:                             k ← kernel[filter][channel][kline][kcol]
11:                             i ← input[channel][kline][kcol]
12:                             acc += ← k × i
13:                         end for
14:                     end for
15:                 end for
16:                 if acc < 0 then
17:                     acc = 0                               ▷ ReLU
18:                 end if
19:                 output[filter][line − 1][column − 1] ← acc
20:                 acc ← 0
21:             end for
22:         end for
23:     end for
24:     return output
25: end function
```

4.1.2 BatchNormalization

There are 6 Batch Normalization layers in the microfaune_ai model after each Conv2D layer normalizing their output. As explained in section 2.2.2, Batch Normalization layers can merge their weights with the previous Conv2D. However, to facilitate the transition of the replicated model from Python to C to FPGA, this layer was replicated during the Python to C keeping the replicated model as close to the original. After quantization, this merge was completed and the Batch Normalization layers were removed from the quantized model in inference.

The pseudo-code for the Batch Normalization layer can be found in algorithm . It iterates through all 64 channels, 431 lines, and columns of the previous layer, applying the calculations outlined in equation 2.2 during inference.

Algorithm 2 BatchNormalization

1: **function** BNORM($input, columns, gamma, beta, mean, variance$)
2: **for** $channel \leftarrow 0$ to $(64 - 1)$ **do**
3: **for** $line \leftarrow 0$ to $(431 - 1)$ **do**
4: **for** $col \leftarrow 0$ to $(columns - 1)$ **do**
5: $valueSqrt \leftarrow sqrt(variance[channel] + EPSILON)$
6: $in \leftarrow input[channel][line][col]$
7: $normalized \leftarrow (in - mean[channel])/(valueSqrt)$
8: $out \leftarrow (gamma[channel] \times normalized) + beta[channel]$
9: $output[channel][line][col] \leftarrow out$
10: **end for**
11: **end for**
12: **end for**
13: **return** $output$
14: **end function**

4.1.3 MaxPool2D

MaxPool2D layers are positioned after every two Conv2D, reducing the number of columns of the model by half, starting with 40 columns and reducing to 5 columns after the last MaxPool2D. The number of lines remains the same since the pooling window is 1 line by 2 columns.

The algorithm 3, shows the pseudo-code of the replicated MaxPool2D layers.

The MaxPool2D algorithm, iterates the 64 channels, 431 lines, and the columns provided. When iterating a line, it compares 2 values simultaneously following the pooling window, keeping the highest value as the output. The output will have 64 channels, 431 lines, and half of the columns of the input to this layer.

Algorithm 3 MaxPool2D

```
1:  function MAXPOOL2D(input, inColumns)
2:      for channel ← 0 to (64 − 1) do
3:          for line ← 0 to (431 − 1) do
4:              outColumn ← 0
5:              for col ← 0 to (columns − 1) increment 2 do
6:                  in ← input[channel][line][col]
7:                  inNext ← input[channel][line][col + 1]
8:                  if in > inNext then
9:                      maxVal ← in
10:                 else
11:                     maxVal ← inNext
12:                 end if
13:                 output[channel][line][outColumn + +] ← maxVal
14:             end for
15:         end for
16:     end for
17:     return output
18: end function
```

4.1.4 ReduceMax

The microfaune_ai model has two ReduceMax layers, one at the end of the CNN section, and as the last layer of the model. It reduces the number of lines or columns to a single value, keeping the highest found as the output. The first ReduceMax reduces the 5 columns produced by the last MaxPool2D to 1 column, and the second ReduceMax reduces the 431 lines to 1 line, as presented in model figure 3.6.

Algorithm 4, shows the pseudo-code of the first ReduceMax and algorithm 5 the second.

The first ReduceMax receives 64 channels, 431 lines, and 5 columns. It is applied to the columns, reducing them from 5 columns to 1 column. Later during development, this ReduceMax layer was removed and the last MaxPool2D changed from a window of 1 line by 2 columns, to 1 line by 10 columns, effectively combining the MaxPool2D with this ReduceMax layer.

The second ReduxeMax receives 431 lines and 1 column. It is applied to the lines, reducing them from 431 lines to 1 line, a single value. This value is the Global Score, the value that will tell us if there is a presence of a bird or not in the input of the model.

Algorithm 4 ReduceMax - 1st

```
1:  function REDUCEMAX_1(input)
2:      for channel ← 0 to (64 − 1) do
3:          for line ← 0 to (431 − 1) do
4:              maxVal ← 0                      ▷ after ReLu activation, min value is 0
5:              for col ← 0 to (5 − 1) do
6:                  value ← input[channel][line][col]
7:                  if value > maxVal then
8:                      maxVal ← value
9:                  end if
10:                 output[channel][line][col] ← maxVal
11:             end for
12:         end for
13:     end for
14:     return output
15: end function
```

Algorithm 5 ReduceMax - 2nd

```
1: function REDUCEMAX_2(input)
2:     maxVal ← 0                      ▷ after sigmoid activation, min value is 0
3:     for line ← 0 to (431 − 1) do
4:         value ← input[line][0]
5:         if value > maxVal then
6:             maxVal ← value
7:         end if
8:         output[line][0] ← maxVal
9:     end for
10:    return output
11: end function
```

4.1.5 GRU

Microfaune_ai model uses two GRUs with each one wrapped inside a Bidirectional layer. The Bidirectional layer provides the wrapped layer with the same input in a forward and backward direction.

TensorFlow has two versions of the GRU implementation, GRUv1 and GRUv2, our model uses the GRUv2 implementation. This implementation was found under the installed Keras directory, inside the file `gru.py`. The listing 4.1 shows the execution path used by the GRU Cells. This code can also be found in the Keras GitHub repository[1].

```
# Taken from 'gru.py', Keras library
 def step(cell_inputs, cell_states):
    """Step function that will be used by Keras RNN backend."""
    h_tm1 = cell_states[0]

    # inputs projected by all gate matrices at once
    matrix_x = backend.dot(cell_inputs, kernel)
    matrix_x = backend.bias_add(matrix_x, input_bias)

    x_z, x_r, x_h = tf.split(matrix_x, 3, axis=1)

    # hidden state projected by all gate matrices at once
    matrix_inner = backend.dot(h_tm1, recurrent_kernel)
    matrix_inner = backend.bias_add(matrix_inner, recurrent_bias)

    recurrent_z, recurrent_r, recurrent_h = tf.split(
        matrix_inner, 3, axis=1
    )
```

[1]Keras, GRU Cell Python code: https://github.com/keras-team/keras/blob/94a1712027366cd626df4f1d77f25f918a18f456/keras/layers/rnn/gru.py#L969

```
################ STEP START ################ None
kernel = [[0 0 0]]
recurrent_kernel = [[0 0 0]]
bias = [[1 1 1]
 [1 1 1]]
recurrent_bias = [1 1 1]
cell_inputs = [[0]]
cell_states = ([[0]],)
------------------------------------------- None
h_tm1: [[0]]
matrix_x (dot) = [[0 0 0]]
matrix_x (bias_add) = [[1 1 1]]
tf.split = [[[1]], [[1]], [[1]]]
x_z = [[1]]
x_r = [[1]]
x_h = [[1]]
matrix_inner (dot) = [[0 0 0]]
matrix_inner (bias_add) = [[1 1 1]]
recurrent_z = [[1]]
recurrent_r = [[1]]
recurrent_h = [[1]]
x_z + recurrent_z = [[2]]
z = [[0.880797088]]
x_r + recurrent_r = [[2]]
r = [[0.880797088]]
x_h + r * recurrent_h = [[1.88079715]]
hh = [[0.954562962]]
h = [[0.113786682]]
################ STEP END ################ None
1/1 [==============================] - 0s 424ms/step
```

Figure 4.1: Print of each value of the GRU for the simple model.

```
19      z = tf.sigmoid(x_z + recurrent_z)
20      r = tf.sigmoid(x_r + recurrent_r)
21      hh = tf.tanh(x_h + r * recurrent_h)
22
23      # previous and candidate state mixed by update gate
24      h = z * h_tm1 + (1 - z) * hh
25      return h, [h]
```

Listing 4.1: Keras GRUv2 Python implementation.

To debug Keras models, a tensor must be added to print when the model is executed. This is done using `backend print_tensor`, providing the tensor we want to see the value and a message describing the value for easy identification among the prints in the console. This is demonstrated in code 4.2 in lines 5 and 7. Adding similar prints to all the GRU cell variables, we get a console print of each step like in figure 4.1. Note that the values shown in that figure are from a small model for debugging and do not reflect a microfaune_ai model execution. These prints do not show when the model is executed on the GPU, so it is required to force TensorFlow to use CPU[2].

```
1 CUDA_VISIBLE_DEVICES=""  # force CPU by not showing available CUDA devices
2 ...
3 # inputs projected by all gate matrices at once
4 matrix_x = backend.dot(cell_inputs, kernel)
5 backend.print_tensor(matrix_x, "matrix_x (dot) =")        # debug dot
6 matrix_x = backend.bias_add(matrix_x, input_bias)
7 backend.print_tensor(matrix_x, "matrix_x (bias_add) =") # debug bias_add
```

Listing 4.2: Printing tensor values using print_tensor.

Figure 4.2 has the diagram of the GRUv2 step function. The microfaune_ai model when reaching the RNN section of the model, the Bidirectional GRUs, has an input

[2] `CUDA_VISIBLE_DEVICES`: https://stackoverflow.com/a/37663502

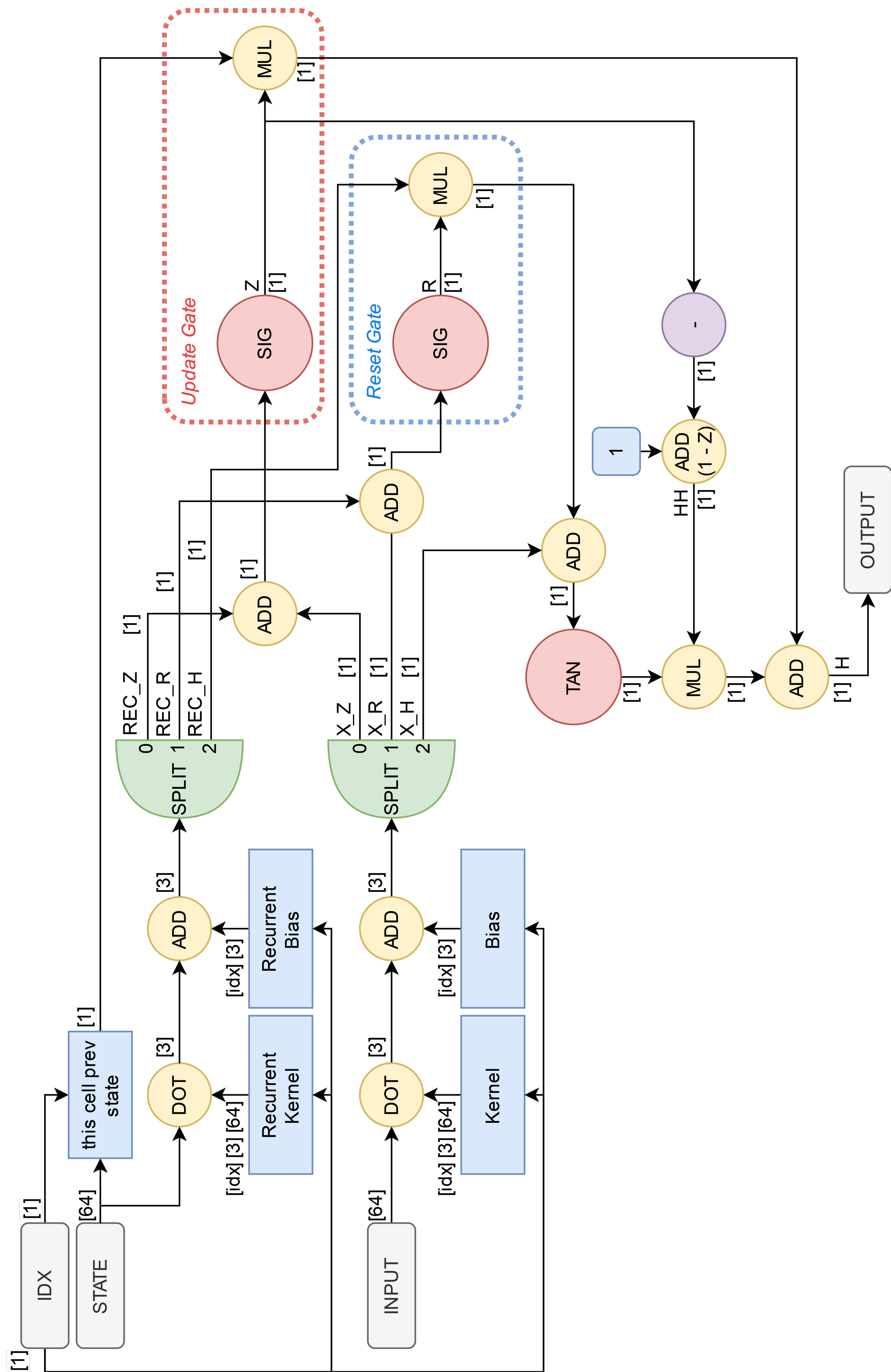


Figure 4.2: Overview of TensorFlow GRUv2 single cell step function.

size of 431 lines and 64 channels. All the GRU cells receive the same input of 1 line with 64 values. The calculation within each cell starts by multiplying the input values by a kernel, resulting in three values. Subsequently, each value is individually added to its respective bias.

In the upper section of the figure 4.2, the recurrent section, the term `STATE` represents the previous state of all 64 GRU cells, the output of the preceding line from all the GRU cells, initialized with zeros at the start of a GRU evaluation. As the evaluation proceeds, these states are continually updated with the output produced by the previous line. The process for `STATE` mirrors what was explained earlier for the input section, with the difference being the usage of recurrent kernel and recurrent bias.

Still, in figure 4.2 and at the input section, the `SPLIT` corresponds to the 3 values calculated earlier. These values are used with the ones from the recurrent section for the Update gate, Reset gate, Sigmoid, and Tanh calculations, previously shown in figure 2.6.

The output is produced, saved internally as a state, and then returned. The diagram has a `IDX`, that contains the index value of the GRU Cell. We have 64 GRU Cells in our model, this index ranges from 0 to 63 inclusive. It is used during the internal state-saving process, so the current cell knows where to place its output value in the state array using its index.

The cells can only read from the state array when all the cells have written to it.

This step function adjusted to our model, is described in the pseudo-code 6. Please note that the `state` is an array shared among all 64 GRU cells, and it can only be updated after all the GRU cells have accessed the current line state for that iteration.

Algorithm 6 GRU cell

```
1: function GRU_CELL(idx, input, kernel, rec_kernel, bias, rec_bias)
2:     for i ← 0 to (3 − 1) do
3:         for column ← 0 to (64 − 1) do
4:             inVal ← input[column]
5:             krVal ← kernel[idx][i][column]
6:             matrix_x[i]+ = (inVal × krVal)
7:         end for
8:     end for
9:     for i ← 0 to (3 − 1) do
10:        matrix_x[i]+ = bias[idx][i]
11:    end for
12:
13:    for i ← 0 to (3 − 1) do
14:        for column ← 0 to (64 − 1) do
15:            inVal ← state[column]
16:            krVal ← rec_kernel[idx][i][column]
17:            matrix_inner[i]+ = (inVal × krVal)
18:        end for
19:    end for
20:    for i ← 0 to (3 − 1) do
21:        matrix_inner[i]+ = rec_bias[idx][i]
22:    end for
23:
24:    z ← SIGMOID(matrix_x[0] + matrix_inner[0])
25:    r ← SIGMOID(matrix_x[1] + matrix_inner[1])
26:    hh ← TANH(matrix_x[2] + (r × matrix_inner[2]))
27:    output ← (z × state[idx]) + ((1 − z) × hh)
28:    state[idx] ← output    ▷ Note: This operation can only be completed after all the
       GRU cells have read the current line iteration state.
29:    return output
30: end function
```

4.1.6 Time-Distrubuted with Dense

Microfaune_ai model has two Time-Distributed Dense layers after the Bidirectional GRUs. The previous layer produces an output of 431 lines by 128 columns. The first Time-Distributed Dense has 64 units, receives this as input, and condenses the 128 to 64 columns. The second has 1 unit, receives the 431 lines by 64 columns, and condenses the 64 to 1 column. After the last Time-Distributed Dense, the 431 values represent the Local Scores of the model, a percentage value from 0 to 1 of the likelihood of a bird sound being present at that timestamp. The Global Score of the model is later calculated in the last model layer, ReduceMax, which gets the single highest value among the 431 values that represent the overall likelihood of a bird sound being present in the entire model input.

The algorithm 7, shows the pseudo-code of the replicated Time-Distributed Dense layers.

The Time-Distributed Dense algorithm iterates the 431 lines and the columns provided, applying the Dense during this processing.

Algorithm 7 Time-Distributed_Dense

```
1: function TIME-DISTRIBUTED_DENSE(input, outCols, kLines, kCols, kernel, bias)
2:     for line ← 0 to (431 − 1) do
3:         for col ← 0 to (outCols − 1) do
4:             acc ← bias[col]
5:             for kcol ← 0 to (kCols − 1) do
6:                 k ← kernel[col][kcol]
7:                 i ← input[line][kcol]
8:                 acc += k × i
9:             end for
10:            output[line][col] ← SIGMOID(acc)
11:        end for
12:    end for
13:    return output
14: end function
```

4.2 Conclusion

This chapter presented the process of each layer in the microfaune_ai model to C. It was shown how to extract the weights from the Python model to be used in the C implementation and later on the FPGA. Also, the extraction of layer outputs was explained making it possible for cross-validating against the original model in Python.

5

HW/SW System for Bird Audio Detection

This chapter presents an overview of the proposed system architecture, the description of each IP core, the software design, and the architecture implementation.

5.1 HW/SW System Architecture

The proposed system architecture has two main blocks (IPs): one that implements the convolutional layers with max pooling and another that implements the Bidirectional GRU layers. The blocks are configurable to support different configurations of the layers.

The Time-Distributed Dense layers and the last Reduce Max of the model are processed in software. As previously explained, the Time-Distributed Dense layers could not be quantized using QKeras, opting to process them on CPU using floats. This processing follows the algorithm explained in sections 2.3.3 and 4.1.6. The last Reduce Max layer of the model only processes a small 431 by 1 vector, so it is also processed on CPU since the performance impact is negligible.

The original microfaune_ai model returns 2 outputs, the Local Score and the Global Score. The Local Score is the score given to each line of the input, while the Global Score is the highest score found among the Local Scores. However, the software implementation proposed only returns the Global Score of the microfaune_ai model for

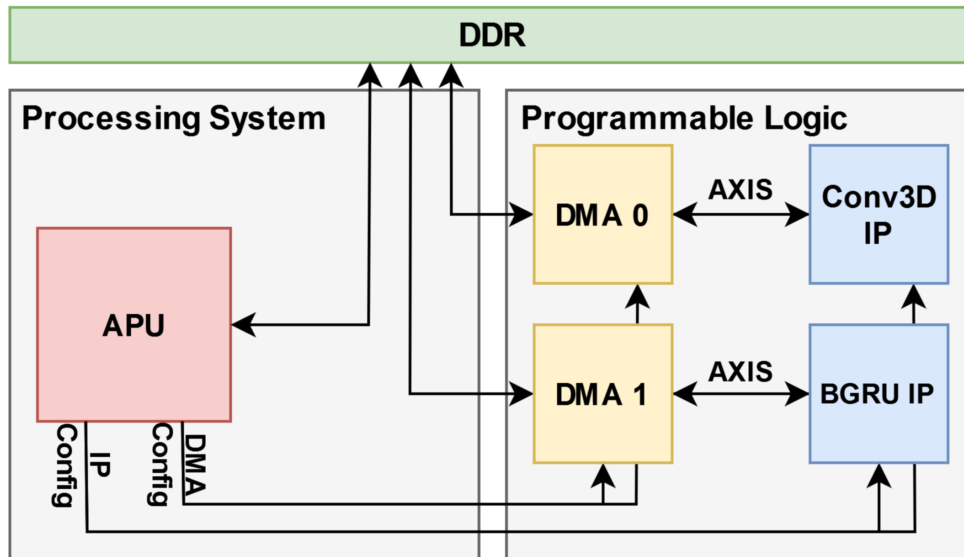


Figure 5.1: Proposed system architecture.

simplification. The Global Score is a number between 0 and 1, inclusive. If it's below 0.5, no bird vocalization was detected, if it's 0.5 or higher, a bird vocalization was detected. The closer the score is to 1, the more confident the model is in its assessment.

Figure 5.1, presents an overview of this proposed architecture.

The system consists of a processor and a hardware accelerator. The accelerator includes two main blocks to process the main layers of the model and two DMA (Direct Memory Access) modules to transfer data between external memory and the cores, one for each core.

The processor not only executes the last layers, but also controls the execution flow of the model and configures the Direct Memory Access (DMA) blocks and the IP blocks.

5.2 IP Blocks

This section shows the developed IP Blocks, how they work and interact within the model, the optimizations, and expected resource usage.

5.2.1 Conv3D IP

The Conv3D IP block implements the Convolution and MaxPool layers. This block can do all 6 Convolutions, 3 MaxPools, and 1 ReduceMax, present in the CNN section of the model.

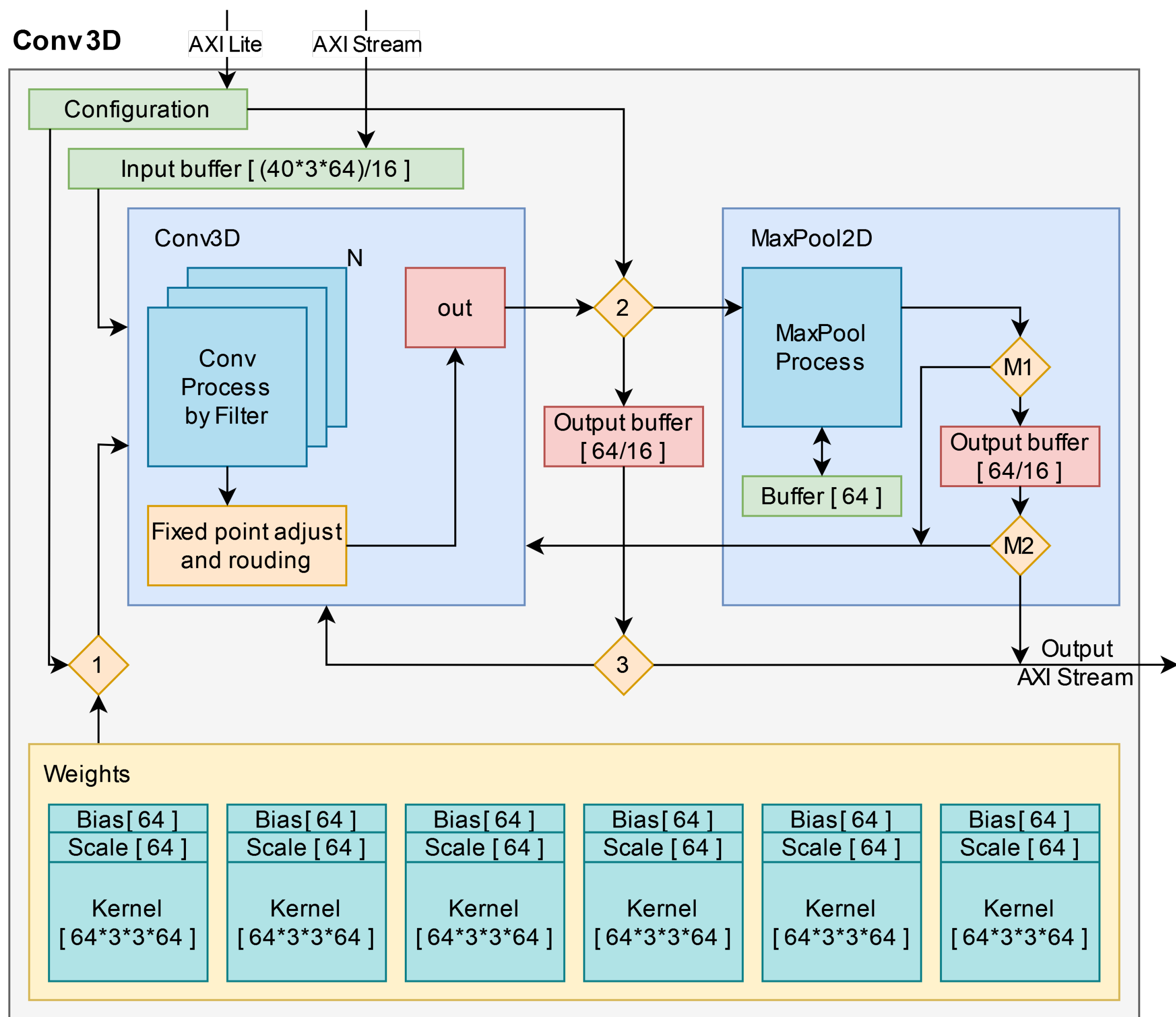


Figure 5.2: Conv3D IP overview

Figure 5.2, presents an overview of the Conv3D IP, where 5 main sub-blocks can be seen.

The Input Buffer sub-block stores data from the input stream. It has enough storage for three lines of the input map. The Conv3D sub-block is responsible for processing the input according to the Convolution algorithm, rounding the output value, and sending it to the next sub-block, either the Output Buffer of the MaxPool2D sub-block or the output buffer. The MaxPool2D sub-block is responsible for executing the Max Pooling layer to the output of the Conv3D sub-block. The Output Buffer sub-block is used to temporarily store lines of the output map before being sent over the AXI-Stream to be stored in external memory. Lastly, the Weights block represents the local memories used to store the weights for all 6 Convolution layers.

The dataflow of this Conv3D IP starts by receiving the layer Configuration and the

first 3 lines, for the 64 channels, of the input feature map. The Weights are saved in the IP memory, and the appropriate set of weights, bias, kernel, and scale are selected based on the layer Configuration received via AXI-Lite. The sub-block Conv3D with the input and the set of weights received, starts processing according to the Convolution algorithm, rounding the output value and sending it to the next sub-block. The next sub-block depends on the layer Configuration, if the current layer should do MaxPooling, the output is sent to the MaxPool sub-block, otherwise, the output is sent to the Output Buffer sub-block. The MaxPool2D sub-block receives the output from the Conv2D sub-block, processes it, and places the result in an internal Output buffer. In case this layer does not have MaxPooling, the output value from the Conv3D sub-block is placed on the Output buffer, organizing the output to be sent over AXI-Stream to the external memory.

The first layer of the microfaune_ai model receives an input of 431x1x40 (the audio spectrogram). The input maps of the following convolutional layers are 431x64x40, 431x64x20, or 431x64x10, as shown in figure 3.6. To implement this Conv3D IP, three options were considered:

1. Create two distinct Conv3D IPs, one for the first layer with an input of 431x1x40, and another for the remaining convolutional layers;
2. A single Conv3D IP that checks if it is the first layer during runtime, and accepts the input accordingly;
3. A single Conv3D IP that runs all layers by padding the input with the necessary channels to 64 channels before calling the model, essentially converting the input to 431x64x40 by only filling the first channel with the valid input. For this to work, the kernel that is 3x3x1x64 (kernel lines, kernel columns, channels, filters), should also be padded to 3x3x64x64 with zeros the same way as the input.

The third option was the quickest to implement in terms of development time. It only required kernel padding on the first layer during weights extraction in Python. The input was also padded during the input feature extraction in Python.

Figure 5.3 shows a visualization of the input on the first channel in green and in red the other channels from index 1 to 63 filled with zeros. The same process is done for the first layer kernel weights channels, 1st channel with the actual kernel values and the rest filled with zeros. This simpler and consistent implementation across all the convolution layers was preferred, even though it increases memory usage to account for the extra padded kernel weights. In case the performance would be an issue, this implementation could be revised.

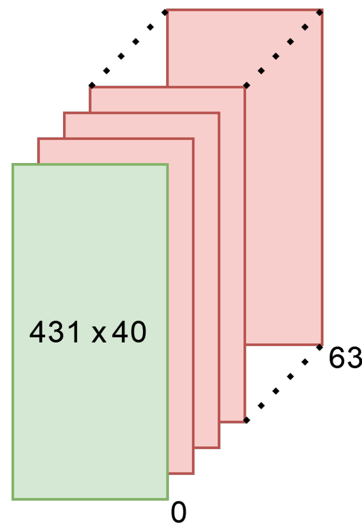


Figure 5.3: FPGA input padding visualization.

Bytes	Conv3D_0	Conv3D_1	Conv3D_2	Conv3D_3	Conv3D_4	Conv3D_5	
Bias	128	128	128	128	128	128	
Kernel	18432	18432	18432	18432	18432	18432	
Scale	32	32	32	32	32	32	
Total	18592	18592	18592	18592	18592	18592	111552

Table 5.1: Bytes used by Conv3D weights.

The weights were embedded into the IP to reduce the number of transactions done within the AXI-Stream, improving performance at the cost of increased memory usage. Table 5.1 shows the memory usage used by the weights in bytes. The quantized weights and the bias use half a byte, as explained in Chapter 3. Even though bias was quantized to 4 bits, it uses 16 bits per value in memory to guarantee a statically aligned fixed-point value that initializes the accumulator. There are 64 bias per layer with each value taking 2 bytes. The kernel takes half a byte per value, being 64 filters, 64 channels, 3 lines, and 3 columns. Additionally, there is one scale of the fixed-point value per filter, each occupying half a byte.

The microfaune_ai model ends the CNN section with a MaxPool2D of 1 by 2, followed by a ReduceMax of 5, changing the output from 10 to 5, and then to one column, shown in figure 3.6. To improve the IP processing, these 2 layers were merged into a single layer, producing a MaxPool that processes 1 by 10 achieving the same result more efficiently.

The process of executing a CNN layer starts by receiving an integer by AXI-Lite indicating the Convolution index in the microfaune_ai model, ranging from 0 to 5. This index tells the core which weight memory to use, if this layer should be proceeded by a MaxPool layer and the size of the maxpool window.

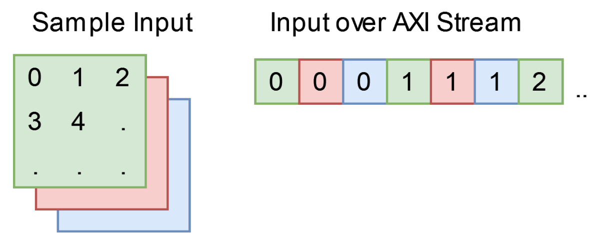


Figure 5.4: Small example of input sent over AXI-Stream.

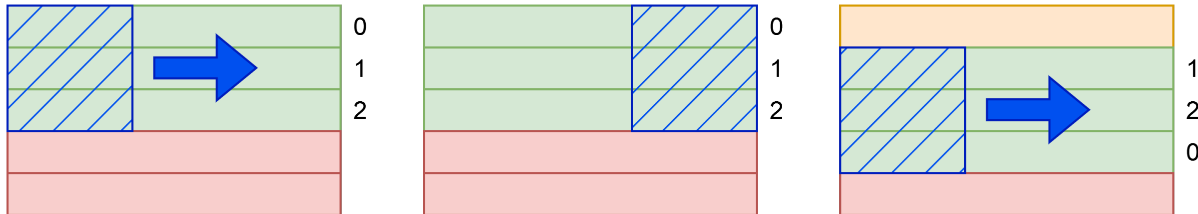


Figure 5.5: Small example of the circular input buffer.

The Input consists of values of 4 bits ordered by the following prioritization: filter, column, line, as illustrated in Figure 5.4.

The first 3 lines of the 64 channels are read from the input AXI-Stream to a circular Input buffer. During the convolution process, when one line is completely processed, the next line is read from the Input and placed into the Input buffer. Figure 5.5, presents a simplified convolution showing the loaded inputs in green, in red the next to be read, and in orange the unloaded input. The values next to the lines indicate the index in the Input buffer of the loaded line.

The Convolution Process executes the algorithm 1, running 16 multiply-accumulations in parallel in each iteration. After the calculation of a full convolution, the value must be adjusted to match the 4-bit fixed-point representation of the output activations (Q0.4). This is achieved by shifting the accumulator value, (scale-1) times to the right. If the value is negative then it is set to 0 (ReLU activation function). The resulting value is rounded to the nearest even, the same as in QKeras.

At decision point 2 (Figure 5.2), if the configuration tells that the current layer does not have MaxPool, the output value is placed on the output buffer. If the output buffer is full, it is sent over AXI-Stream. Otherwise, the Conv3D must produce more output values to fill the buffer before sending the data to external memory.

When the convolutional layer is followed by a maxpool layer, the output map must be processed by the MaxPool block before being sent over AXI-Stream. The MaxPool module receives and processes the input according to the algorithm 3. MaxPool2D uses an internal buffer to track the highest value it has found so far. At decision point M1, it

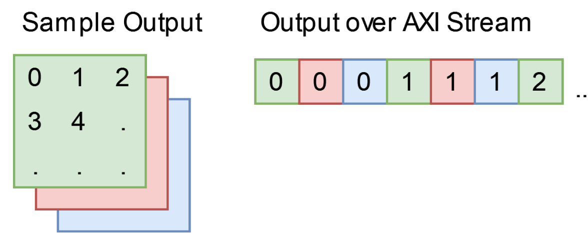


Figure 5.6: Small example of output sent over AXI-Stream.

checks if the number of required values compared in the current MaxPool was reached. If not, the convolution must produce more values, otherwise the highest value is placed into the output buffer. Moving to decision point M2, similarly to decision point 3, it is checked if the output buffer is full so it can send over AXI-Stream, otherwise more output values must be produced.

The Output of Conv3D IP consists of values of 4 bits ordered by the following prioritization: filter, column, line (See Figure 5.6).

The optimization strategy explored the parallelism within the convolution accumulation calculations during kernel iteration cycles 1. This was achieved by unrolling the cycles in the convolutional algorithm.

The unroll factors selected for kernel cycles were 1 and 3, representing no cycle unrolling and full cycle unrolling, respectively, appropriate for a 3 by 3 kernel. Factors 1, 2, and 4 were explored in the accumulation calculation cycle. Given that each cycle already handles 16 calculations in parallel, the unroll factors explored were: 1 no unrolling, 2 doubling to 32 operations per cycle, and 4 quadrupling to 64 operations per cycle, consistent with the total number of filters.

Table 5.2, shows the explored unroll settings for each cycle, as well as the expected total number of cycles and FPGA resources usage reported by HLS C synthesis.

This study targets the Ultra96-V2 platform. The *Kernellines* and *Kernelcolumns* represent the selected unroll factor for kernel iteration, and 16*acc* the accumulation cycles with the respective selected unroll factor. Note that cycles are the expected number of cycles to execute the layer with higher computational complexity, that is, the second layer with an input of 431x64x40, kernel 3x3, 64 filters, and a MaxPool of 1x2. The Baseline design is the solution without unrolling and a parallelism factor of 16 inside the accumulation cycle. The number of cycles superior to the Baseline is marked in bold. The resource usage superior to the target platform is also marked in bold.

One of the explored situations was unrolling all the cycles, this being the Fastest option when looking at the number of cycles. This option exceeds the FPGA resources

Conv3D Unroll	**K. lines**	**K. cols**	**16 acc**	**Cycles**	**BRAM**	**DSP**	**FF**	**LUT**
Ultra96-V2	-	-	-	-	216	360	141120	70560
Baseline	1	1	1	52961606	79	0	881	7161
	1	1	2	33101126	79	0	885	8017
Selected	1	1	4	22067526	150	0	1138	9392
	1	3	1	**101509446**	79	0	1087	9337
	1	3	2	**84959046**	79	0	1088	12179
	1	3	4	15447366	**458**	0	1618	16256
	3	1	1	**62891847**	79	0	1104	9700
	3	1	2	43031367	79	0	1062	12604
	3	1	4	33101126	150	0	1803	17519
	3	3	1	**95992646**	79	0	1519	16561
	3	3	2	**76132167**	79	0	1533	25153
Fastest	3	3	4	1103696	**1422**	0	2778	38398

Table 5.2: Conv3D IP cycle unrolling cycles exploration.

by around 6 times the available BRAMs. The architecture selected for implementation was the one obtained by unrolling only the most inner cycle changing the accumulation parallelism from 16 to 64. This was the fastest option within the target platform resource limits.

5.2.2 BGRU IP

The second hardware block, called BGRU IP, implements the Bidirectional GRU layers. This block can do both Bidirectional GRUs present in the RNN section of the model. Bidirectional GRU is achieved by applying the GRU layer to an input in a forward direction, and then the same input in a backward direction, ending with the merge of both outputs.

Figure 5.7, presents an overview of the BGRU IP, where 4 main sub-blocks can be seen.

The Input Buffer sub-block stores the entire feature map returned by the previous layer: 431×64 on the first Bidirectional GRU layer and 431×2 on the second. The Bidirectional GRU sub-block is responsible for iterating the input according to the GRU algorithm and managing the GRU Cell sub-block. The GRU Cell sub-block receives the input line from its manager, processes it, and then returns the output to its manager, the Bidirectional GRU sub-block. Lastly, the Weights block represents the memory of weights of both Bidirectional GRU layers.

The dataflow of the BGRU IP starts by receiving the layer Configuration and the entire input feature map. The Weights are saved previously in the IP memory, and the appropriate set of weights, bias, recurrent bias, kernel, and recurrent kernel are selected

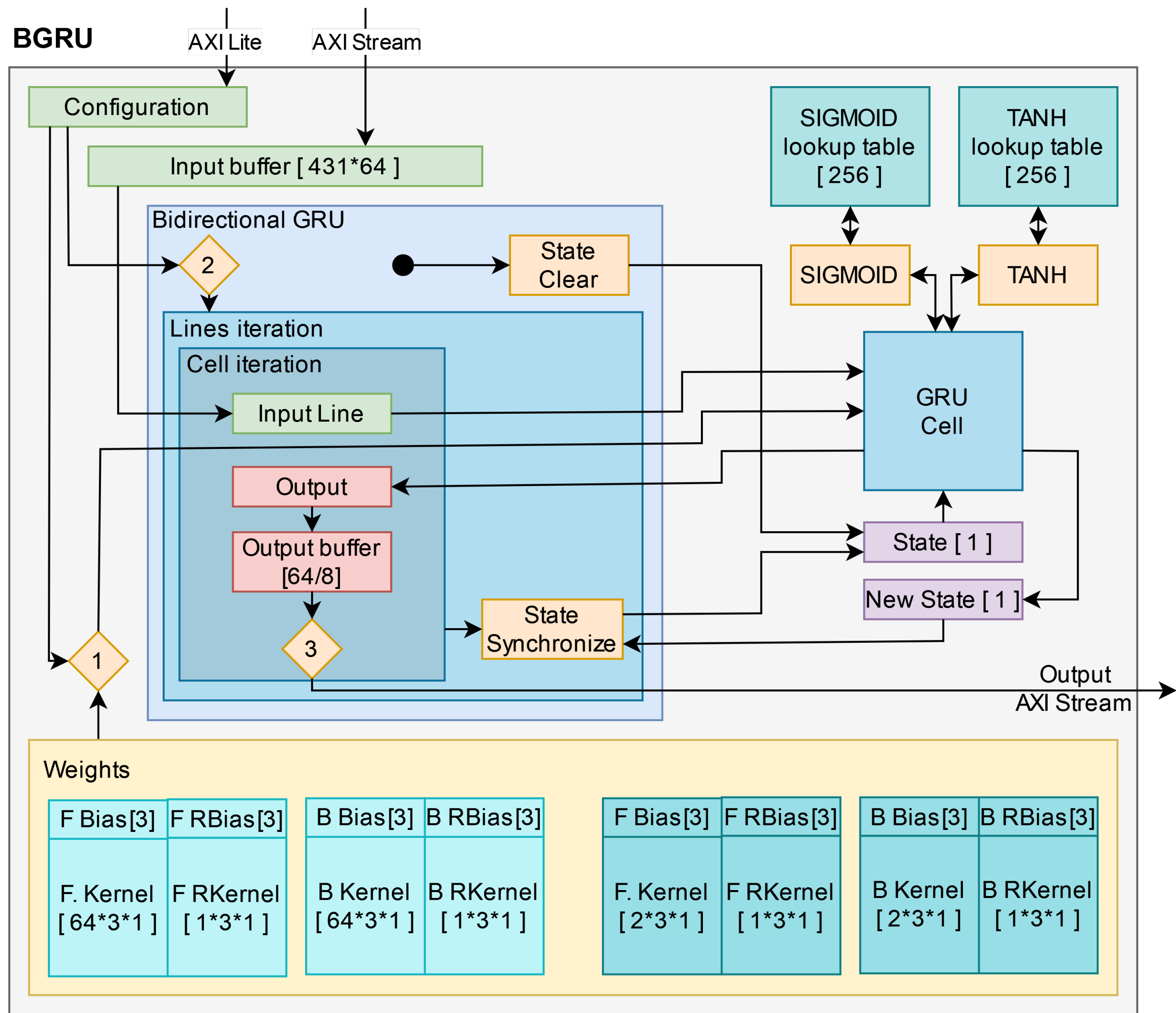


Figure 5.7: BGRU IP overview

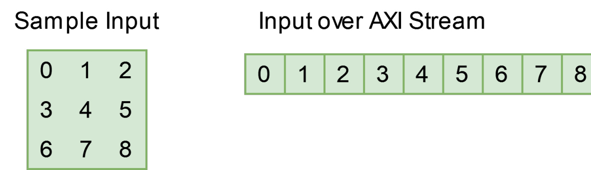


Figure 5.8: Small example of an input sent in a Forward direction over AXI-Stream .

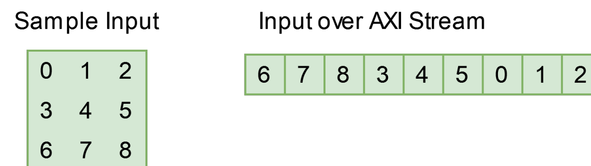


Figure 5.9: Small example of an input sent in a Backward direction over AXI-Stream .

based on the layer Configuration received via AXI-Lite. The sub-block Bidirectional GRU clears the previous GRU state and with the input and the set of weights uses the GRU Cell sub-block to process the input map. The GRU Cell sub-block receives a line for processing from the Bidirectional GRU sub-block. Both Sigmoid and Tanh functions are used in the calculation of the GRU cell, Before returning the output, the module updates the GRU state with that output. The output produced by the GRU Cell block is received by the Bidirectional GRU sub-block, which places this output value in the Output buffer to be sent over AXI-Stream when full.

The Input consists of values of 8 bits ordered by the following prioritization: line, column. Since GRU layer is inside a Bidirectional wrapper, the Forward and Backward inputs have a small ordering difference. Inputs in a forward direction start on the first line on the first column and iterate by column and then lines. Inputs in a backward direction start on the last line, but the first column on that same line, iterating lines in a backward direction but the column in a forward-like direction.

Figure 5.8 shows a small example when sending an Input in a Forward direction and figure 5.9 in a Backward direction.

Similarly to the Conv3D IP, the weights were also embedded into the IP, increasing performance at the cost of increased memory usage. Based on the extracted weights, as explained in Chapter 4, each weight was quantized by truncation and uses one byte. Table 5.3 shows the memory usage used by the weights in bytes.

There are 3 bias and 3 recurrent bias per cell and direction. The kernel is based on the number of input columns times 3 times the number of cells. BGRU_0 has 64 input columns and BGRU_1 has 2 input columns. For the recurrent kernel, the same applies but instead of input columns, it is based on the number of cells, in this case, there is

Bytes	BGRU_0	BGRU_1	
Forward Bias	3	3	
F. Recurrent Bias	3	3	
F. Kernel	192	6	
F. Recurrent Kernel	3	3	
Backward Bias	3	3	
B. Recurrent Bias	3	3	
B. Kernel	192	6	
B. Recurrent Kernel	3	3	
Total	402	30	432

Table 5.3: Bytes used by BGRU weights.

only 1 cell.

The process of executing a Bidirectional GRU layer starts by receiving an integer by AXI-Lite that represents the index of the GRU in the microfaune_ai model. This configuration index ranges from 0 to 3. Indexes 0 and 1 are related to the first BGRU, and indexes 2 and 3 to the second BGRU. At decision point 2 (See Figure 5.7), this index determines the direction of the Bidirectional wrapper, even numbers define the forward direction, while odd numbers define the backward direction. Also, similar to the Conv3D IP, this configuration index determines the set of weights to be used.

The first Bidirectional GRU receives an input of 431x64, and the second 431x2. This input is fully read from AXI-Stream and stored in an Input buffer. This approach was chosen instead of reading the AXI-Stream as needed because all the GRU cells read the same input. It is true that this model only uses 1 GRU cell, but this makes supporting more GRU cells easier if needed.

The Bidirectional GRU block starts clearing all the previous GRU cells state, setting them to zero. Then it starts iterating input lines, and starting the GRU cell with the current line.

The GRU cell executes the algorithm 6 previously described. It uses a custom Sigmoid and Tanh, each with its custom lookup tables designed for this model. Each of these lookup tables uses 256 bytes and is suggested to recreate them if the model is retrained since the range of input values might change.

Each one of the GRU cells has a filter index assigned, and based on this they place their calculated output into the New State array using their index. Since this model only uses 1 cell, this array only has 1 position. The algorithm ends by setting this New State and returning a single output value.

The output buffer is filled with the output returned by the GRU cells and sent over

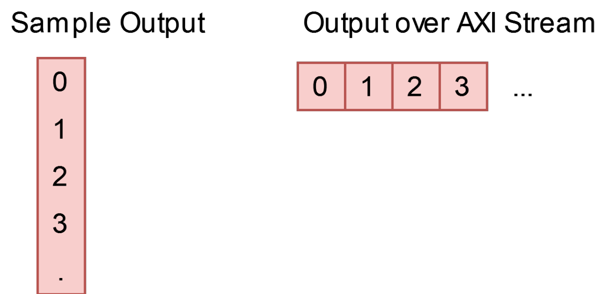


Figure 5.10: Small example of output sent over AXI-Stream.

Conv3D Unroll	Cycles	BRAM	DSP	FF	LUT
Ultra96-V2	-	216	360	141120	70560
BGRU IP	186633	22	0	1438	10045

Table 5.4: BGRU IP expected FPGA resource usage.

AXI-Stream back to the external memory when this buffer is full. The Output consists of values of 4 bits ordered by the following prioritization: line, column (See Figure 5.10).

The output returned is a GRU output, but the required output is a Bidirectional GRU output, a merge of both Forward and Backward directions. This merge is done via post-processing, as explained in section 5.3.2.

When the cell iteration ends, State Synchronize is called which sets New State as the current cell State.

The Bidirectional GRU process continues for the subsequent input lines, ending when all the lines of the input have been iterated.

Parallelism was not explored because the implemented model only uses one GRU cell. Using more cells, it would be possible to process them in parallel due to their processing being independent between cells, while only being dependent on the previously calculated State.

The developed BGRU IP was not subjected to the same optimization level as the Conv3D IP counterpart. This decision was because the number of expected cycles was much lower than the Conv3D IP, and the expected FPGA resource usage was well within the target platform limits. Also, since the model only uses one GRU cell, exploring the parallelism of GRU cells is not available.

Table 5.4, shows the expected total number of cycles for the worse case, with the first layer receiving an input of 431x64, and the FPGA resources usage reported by HLS C synthesis.

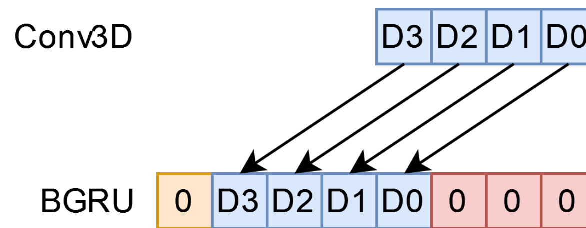


Figure 5.11: Output conversion from Conv3D to BGRU, 4 bits to 8 bits.

5.3 Post-Processing IP Blocks

Post-processing blocks are required at different stages of the model: CNN to RNN, merging Forward and Backward GRU into a Bidirectional GRU output, and lastly RNN to Time-Distributed.

5.3.1 CNN to RNN Post-processing

The Conv3D IP and BGRU IP use different bit width values for input and output. Conv3D IP outputs 431x64 values with 4 bits each, and the first BGRU IP accepts a layer 431x64 with activations represented with 8 bits.

Because of this reason, a conversion between Conv3D_5 and BGRU_0 must be made. Figure 5.11 shows the bit structure of a single value for the Conv3D output and the BGRU input.

To convert the Conv3D to BGRU, the Conv3D value must be shifted to the left by 3 bits filled with zeros, and the 7th bit, the sign bit, is also set to zero. This process is done for every output value of the Conv3D and sent to the BGRU IP. This is in accordance with the fixed-point representation of the input data of the BGRU (Q1.7).

5.3.2 BGRU Output

Since the Bidirectional GRU is a combination of two GRUs, Forward and Backward directions, both outputs must be merged before passing to the next layer: the second BGRU or the Time-Distributed layer.

Figure 5.12, shows a simple example of a BGRU using 2 cells.

Each GRU produced an output of 3 lines and 2 columns. The number of columns produced is equal to the number of cells. To combine the Forward GRU output with the Backward GRU output, both outputs must be interpolated by line. Since this example

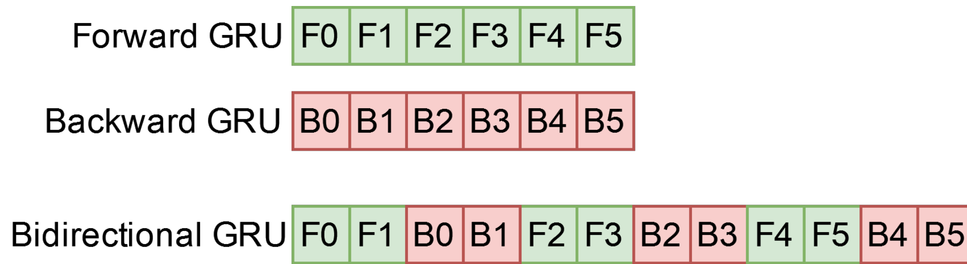


Figure 5.12: Example of BGRU post-process for GRUs with 2 cells.

has 3 lines per GRU and 2 columns per line, the resulting output of the BGRU is the 1st line of the Forward, then the 1st line of the Backward, followed by the 2nd line of the Forward, continuing this process until the last line.

This post-process is done by the CPU of the Processing System.

5.3.3 RNN to Time-Distributed

Similar to the conversion between Conv3D and BGRU, RNN to Time-Distributed requires a conversion from 8 bits to float.

Code 5.1 shows the function that converts a single value from 8 bits to float. It is called inside a cycle that iterates through each output value produced by BGRU_1.

```
1  float fixed81ToFloat(unsigned char cVal) {
2      int negSign = -1 ^ 0xFF;
3      int value;
4      if (cVal & 0x80)                    // check if negative bit is active
5          value = negSign | cVal;
6      else
7          value = cVal;
8      return ((float)value/(float)(128)); // 127 = 2^7 --> 7 fractional bits
9  }
```

Listing 5.1: RNN output to Time-Distributed Dense input

This function starts by creating a variable that will be used to extend the sign bit in case it is negative, *negSign*. Then it checks if the 7th bit is active. If true, extend it, otherwise place the 8-bit value into an integer. This function returns the value divided by 2 to the power of the number of fractional bits, converted to float value.

5.4 Hardware Design

The Conv3D and BGRU IPs were designed using High-Level Synthesis (Vitis HLS) and integrated with the processor using Vivado as illustrated in diagram 5.13. A High Performance (HP) port was used to connect both DMAs to the PS. Each IP is connected to its respective DMA so they can read the input and write the output into the external memory.

5.5 Software Design

The software that implements the microfaune_ai model using the FPGA implemented layers can be described using a state diagram, figure 5.14.

In grey are the functions executed on the CPU, and in blue are the model layers implemented in the FPGA: Conv3D IP, and BGRU IP. The model layers that are executed on CPU using floats are represented in red, and the output at that particular stage in yellow.

The software starts by initializing the AXI-Lite and AXI-Stream present in Conv3D and BGRU IPs.

The Model Predict accepts an input of 431x64x40, with only the 1st of the 64 channels filled, and then Conv3D IP is called multiple times with their respective layer configuration. The input and output passed on between these layers are always read and written to the external memory. The input is sent through the DMA, and the PS then waits for the transaction completion by checking the status via polling. This interaction happens at every Conv3D_X call, ending with Conv3D_5 where it gets the output from external memory.

At Output Conversion A, the first output conversion is executed that converts the CNN output to the RNN output, section 5.3.1.

The BGRU IP is called the first time for a forward pass, and then using the same input in a backward pass. This is not executed in parallel, because we only have one BGRU IP, as explained previously. The output is placed back into external memory, and upon the CPU receiving both forward and backward outputs, it organizes them for the next layer as previously described in section 5.3.2. The BGRU IP is called 2 more times, executing the same job again, but this time using the input from the previous call, BGRU_0.

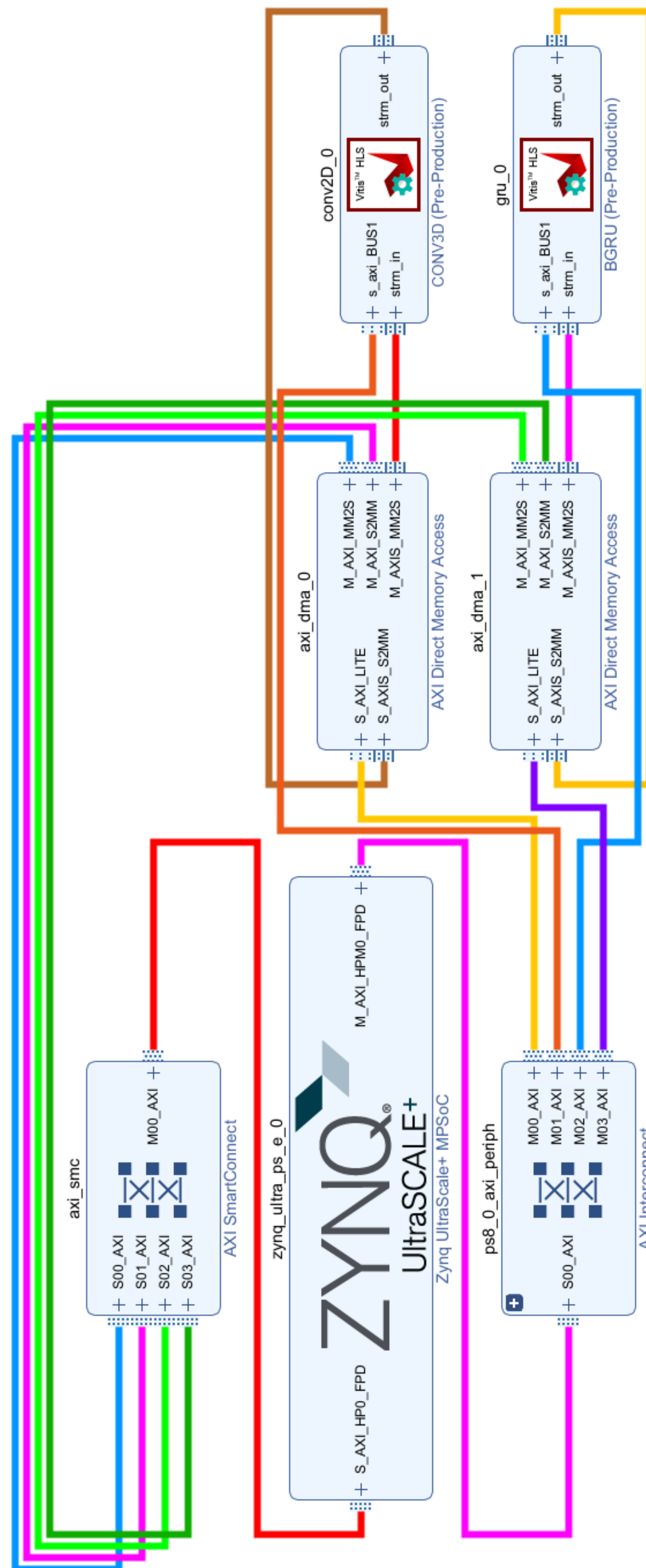


Figure 5.13: Vivado diagram.

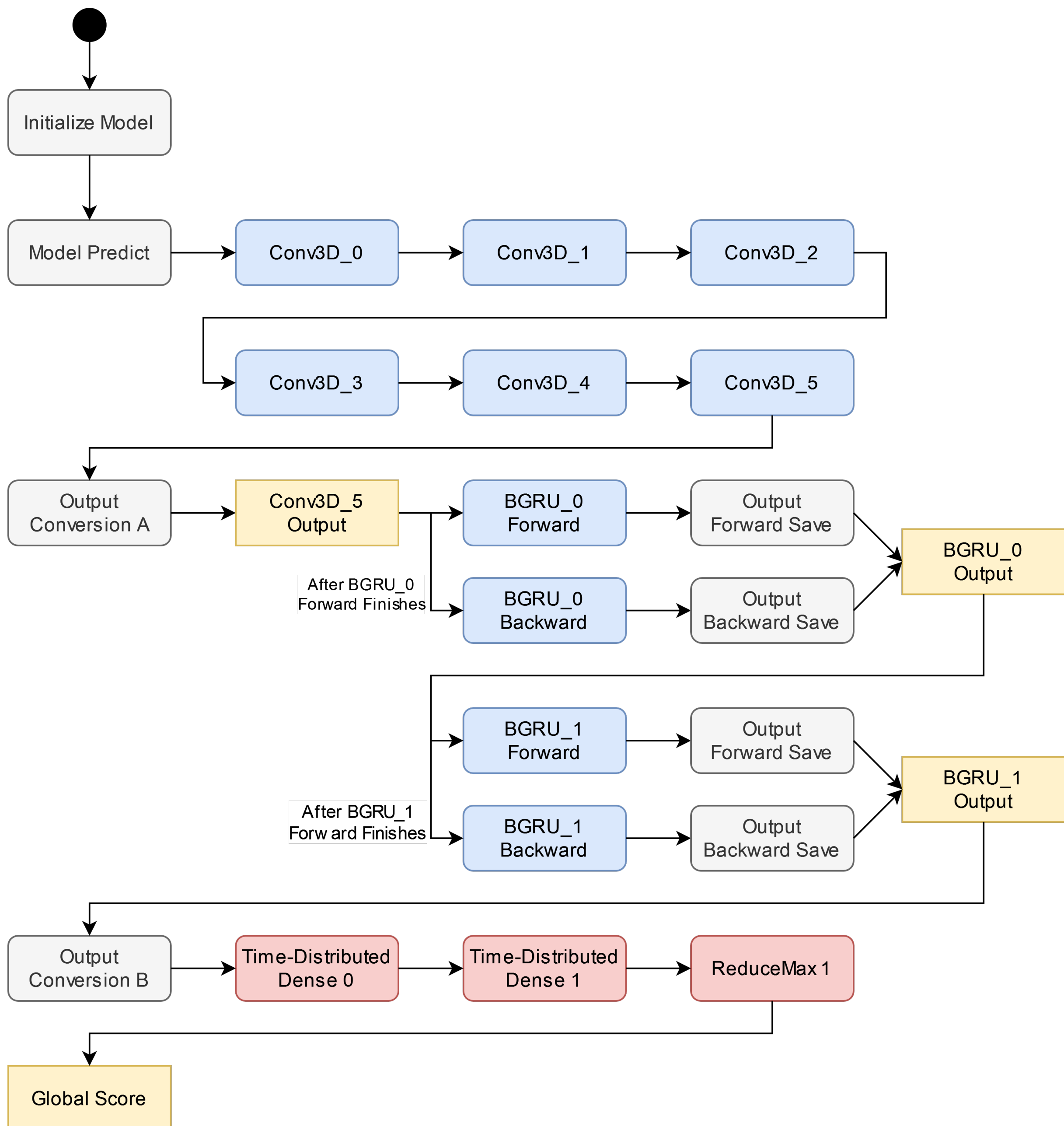


Figure 5.14: State diagram of the software that executes the microfaune_ai model.

The CPU then converts the RNN output to be used in the Time-Distributed Dense layers, section 5.3.3. The model execution ends after the next layers complete their execution, returning a single output float value called Global Score.

5.6 Conclusion

This chapter presented an overview of the proposed high-level system architecture with the interaction between the Processing System and the IP Blocks. The HLS description of each IP core with a detailed explanation of their inner workings. The software design of the model, the software layers, and the data conversions between model sections. Finally, the implementation of the Block design in Vivado was illustrated.

6

Accuracy & Performance Results

This chapter shows the accuracy and performance results of the replicated model in hardware using an FPGA, against the original and quantized models.

6.1 Model Accuracy

We assessed the performance of the QKeras model implemented in hardware on the FPGA and then compared it with the performance of the original microfaune_ai and QKeras models. All tests used 400 samples from the dataset freefield1010, 200 with positive features and 200 with negative features. The performance results were determined by averaging the execution time across the 400 samples, while the resource usage metrics were obtained from Vivado's FPGA resource usage report. The evaluated models were:

- Original microfaune_ai model without any modification, using float.
- Modified model that only uses 1 GRU cell, without quantization, using float.
- Quantized modified model using QKeras. Note that only the CNN part of the model is quantized at 4 bits with 0 integers, while the rest uses float.
- Quantized modified model, with the RNN portion being quantized with truncation 8 bits with 1 integer, running on the FPGA.

	Count		*Percentage*	
	Correct	**Incorrect**	**Correct**	**Incorrect**
Original	359	41	89.75%	10.25%
Modified	345	55	86.25%	13.75%
QKeras	320	80	80.00%	20.00%
FPGA	318	82	79.50%	20.50%

Table 6.1: Accuracy between the Original, Modified, QKeras, and FPGA models.

Weights memory	**Convolutions**	**1st GRU**	**2nd GRU**	**Non-Quantized**	**Total**
Model w/ 64 cells	111552 B	49920 B	74496 B	66052 B	302020 B
Model w/ 1 cell	111552 B	402 B	30 B	66052 B	178036 B
Reduction Factor	1	124.2	2483.2	1	1.7

Table 6.2: Weights memory usage comparison between a model with 64 GRU cells and 1 GRU cell, using the previously mentioned weights calculations.

As shown in the table 6.1, the removal of most of GRU cells impacted the model, reducing its accuracy from 89.75% to 86.25%, 3.5%.

Comparing the FPGA implementation with the QKeras model, the loss in accuracy is 1.5%, indicating that the hardware implementation is working as expected with a small accuracy degradation due to RNN being quantized by truncation to 8 bits.

The same FPGA implementation compared against the Modified model, has a loss of 6.75%. It is a considerable accuracy reduction but expected since the Modified as float precision, and the FPGA implementation was quantized to 4 bits on the CNN and quantized by truncation on the RNN to 8 bits.

The reduction of GRU cells positively impacted the memory usage by the model weights while retaining the capability to detect bird vocalizations. This reduced the model weights memory usage by a factor of 1.7, from 302 KB down to 178 KB, table 6.2. This reduction is important as it is possible to consider smaller and more cost-effective FPGA options when acquiring multiple units for monitoring extensive areas, such as national parks like Yellowstone in the USA.

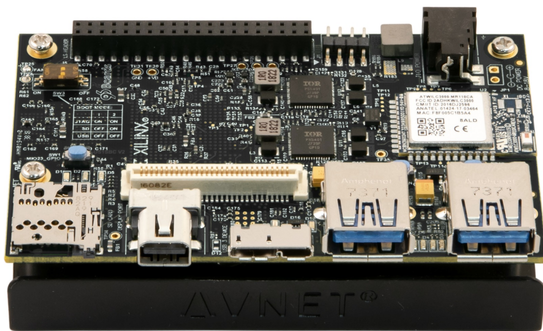
6.2 Performance and Resources

This subsection the performance gathered for each layer running in the FPGA, the pre-processings running on the CPU such as the transition from CNN to the RNN portion of the model, the last non-quantized layers running on CPU, as well as the overall performance and conclusions when comparing to the original model. It is also shown the FPGA resource usage by the developed IPs.

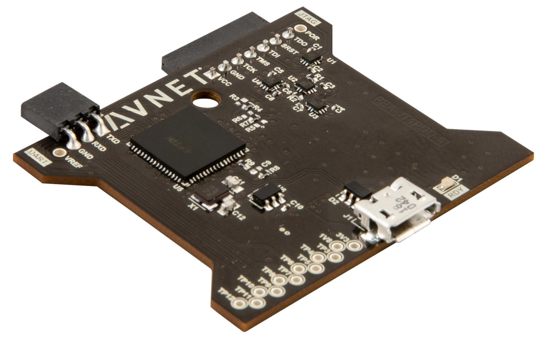
The target platform used was the Avnet Ultra96-V2[1] [2] and contains the Xilinx Zynq UltraScale+ ZU3CG[3] MPSoC. The FPGA resources are as follows:

- **BRAM** = 216
- **FF** = 141120
- **LUT** = 70560
- **DSP** = 360

To program this platform, it is necessary a separate JTAG adapter referred to as AES-ACC-U96-JTAG[4].



(a) Avnet Ultra96-V2



(b) AES-ACC-U96-JTAG

Figure 6.1: Target Platform

6.2.1 CNN Accelerator Performance

To evaluate the performance and resource usage for the CNN part of the model, two scenarios were used. A Baseline without optimizations and the developed Optimized implementation with the optimizations used and explained so far. The previously mentioned table 5.2, shows the exploration to find the Optimized implementation, known as Selected.

Table 6.3, shows the time in milliseconds the FPGA took to process each convolution layer. Notice that Conv_1 takes longer than Conv_0, which is expected because for

[1]Avnet Ultra96-V2: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/

[2]Avnet Ultra96-V2 (AMD product page): https://www.xilinx.com/products/boards-and-kits/1-vad4rl.html

[3]ZU3CG MPSoC: https://docs.amd.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide

[4]AES-ACC-U96-JTAG: https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-acc-u96-jtag-3074457345635355958/

Milliseconds	**Conv 0**	**Conv 1**	**Conv 2**	**Conv 3**	**Conv 4**	**Conv 5**	**Total**
Baseline	497.2	502.0	248.6	251.0	124.3	124.4	1635.5
Optimized	188.3	193.1	94.1	96.6	47.1	47.2	666.4
Speed Up Factor	2.64	2.60	2.64	2.60	2.64	2.64	2.63

Table 6.3: Comparison of the milliseconds used in the CNN portion of the model, between Baseline and Optimized.

FPGA Resources	**BRAM**	**DSP**	**FF**	**LUT**
Baseline	44.5	0	1052	1616
Optimized	85	0	1256	3202
Resources Increment Factor	1.9	1	1.19	1.98

Table 6.4: FPGA resources comparison used in the CNN portion of the model, between baseline and optimized.

every 2 Convolution layers MaxPool is executed. After a MaxPool is executed the number of columns is halved, except on the last Conv_5 which does a MaxPool of 1 by 10.

Optimized had a speed up of 2.64 times when compared to the Baseline, and it is consistent throughout the different Convolution and MaxPool configurations used by this model.

Table 6.4 compares the FPGA resources used between Baseline and Optimized. Optimized uses almost twice the total resources of the Baseline, but it is more than double faster than the Baseline.

By comparing the collected results, we can conclude that optimizations have successfully reduced the evaluation time from 1.648 seconds to 679 milliseconds, a reduction of approximately 1 second, and improved performance by almost 2.46x.

6.2.2 RNN Performance in FPGA

The implemented model's RNN section did not undergo the same level of optimization exploration as its CNN counterpart. This decision was influenced by the already low execution time compared to convolutions. Furthermore, the exploration of parallelism of the GRU cells was limited due to the presence of only 1 GRU cell instead of the original 64 cells, explained in chapter 5.2.2.

Table 6.5 shows the execution time in milliseconds per Bidirectional GRU layer and

Milliseconds	**BGRU 0**	**BGRU 1**	**Total**
Optimized	4.208	0.4396	4.646

Table 6.5: Results of the execution time of both Bidirectional GRUs.

FPGA Resources	**BRAM**	**DSP**	**FF**	**LUT**
Optimized	10	0	1374	3304

Table 6.6: FPGA resources used in the RNN portion of the model.

their combined time. It is not shown the Baseline because no optimization was conducted, but for consistency purposes, the Optimized implementation is the one shown.

Table 6.6 shows the FPGA resource usage by the BGRU IP.

The collected results show that the RNN portion of the model takes around 4.6 milliseconds to complete, substantially faster than the CNN counterpart.

6.2.3 Pre-processing time between CNN and RNN on CPU

Given that the Conv3D IP outputs different bit width values than those acceptable by the BGRU IP, a conversion between them was made. The same goes for between BGRU IP and the Time-Distributed layers ran on CPU.

The performance metrics were taken and can be seen in table 6.7, and indicate that the overall pre-processing execution time is significantly small in comparison to the model layers. Therefore, no IP core was developed for this purpose. No resource usage metrics were taken, since this is done on CPU.

6.2.4 Non-Quantized Layers Times on CPU

The last 3 layers of the model, both Time-Distributed layers and the last Reduce Max, were not quantized and are executed on CPU, section 3.4.

The results can be found in table 6.8, only the Optimized execution times are shown since they are the same as the Baseline. These results show that the execution time of all the CPU layers is relatively small. However, if it becomes possible to quantize these

Milliseconds	**CNN to RNN**	**RNN to T.Dist.**	**Total**
Optimized	0.577	0.038	0.615

Table 6.7: Execution times taken on pre-processing between CNN, RNN, and final layers.

Milliseconds	Time-Distributed 0	Time-Distributed 1	Reduce Max	Total
Optimized	6.220	1.202	0.009	7.431

Table 6.8: Execution times of the final layers running on CPU.

Milliseconds	CNN	CNN to RNN	RNN	RNN to T.D.	CPU Layers	Total
Baseline	1635.5	0.577	4.646	0.038	7.431	1648.2
Optimized	666.4	0.577	4.646	0.038	7.431	679.1
Speed Up Factor	162.9%	100%	100%	100%	100%	159.2%

Table 6.9: Comparison of the number of milliseconds used in different sections of the model, between Baseline and Optimized.

layers and explore the possibility of layer parallelism in the FPGA, there is room for improvement.

6.2.5 Overall Performance and Resources

Aggregating all the previous results from the various tables, we build the table 6.9 that presents the performance between Baseline and Optimized, table 6.10 with the overall percentage of time used by each portion of the model for processing, and table 6.11 that shows the FPGA resource usage.

Comparing the collected results, it is clear that the majority of the execution time is spent on the CNN portion of the model, taking 98% of the execution time. This is primarily because the initial layers of the model handle larger inputs, measuring 431x40x64 for the first and second layers. In contrast, the RNN portion of the model deals with inputs of 431x64 and 431x2 for the 1st and 2nd Bidirectional GRUs, respectively. Despite the lack of parallelism exploration in the RNN, the Optimized implementation is improved compared to the Baseline, doubling the speed and reducing the execution time by over a second. This improvement occurs even though the final layers run on the CPU and there is pre-processing between model sections, also executed on the CPU.

Also, the Optimized implementation on the FPGA was compared against a software

	CNN	CNN to RNN	RNN	RNN to T.D.	CPU Layers	Total
Optimized	666.4 ms	0.577 ms	4.646 ms	0.038 ms	7.431 ms	679.1 ms
% Time Executing	98.13 %	0.08 %	0.68 %	0.006 %	1.09 %	100 %

Table 6.10: Presenting the percentage of execution time across the multiple sections of the Optimized implementation.

FPGA Resources	**BRAM**	**DSP**	**FF**	**LUT**
Conv3D IP	85	0	1256	3202
BGRU IP	10	0	1374	3304
Optimized	101	0	13971	14366
ZU3CG FPGA	216	360	141120	70560
Usage %	46%	0%	9.9%	20.36%

Table 6.11: Optimized resource usage.

Milliseconds	**CNN**	**CNN to RNN**	**RNN**	**RNN to T.D.**	**CPU Layers**	**Total**
ARM	53477.6	0	10.52	0	7.431	53495.3
Optimized	666.4	0.577	4.646	0.038	7.431	679.1
Speed Up Factor	8025%	0%	226%	0%	100%	7877%

Table 6.12: Comparison of the number of milliseconds used in different sections of the model, between ARM CPU and Optimized.

implementation running on the ARM CPU of the Ultra96. This software implementation was created using the C code taken from the Optimized version and replacing the HLS-specific code with C. The AXI-Lite and AXI-Stream interactions were replaced with C pointers for memory access. Then this C code was executed on the Ultra96-v2 ARM CPU and its performance was measured.

Table 6.12 shows the average from multiple runs and the speed-up factor the Optimized FPGA implementation had compared to the full software implementation on the ARM CPU.

This software implementation can be considered an alternative to the Optimized FPGA implementation. However, its performance is considerably lower taking up to 8025x longer to perform an evaluation, taking around 53 seconds, while the Optimized takes 679 milliseconds.

In a real-world scenario where one of these implementations is chosen and considering the model input is 10-second audio maps, the software implementation will stay longer evaluating than listening for a bird. The Optimized FPGA implementation will stay longer listening than evaluating, increasing the chances of catching a bird vocalization.

6.3 Conclusion

In summary, the results of the Optimized FPGA implementation show an accuracy of around 79.50%, 6.75% less than the modified model in Python. Furthermore, optimization has reduced the FPGA's execution time to less than 1 second, which was

an objective to meet. This enables the system to gather data for 10 seconds and execute the evaluation in less than a second, around 6% of the time is used for evaluation, increasing the chances of listening to a bird's vocalization more often.

Despite being around 7 times slower than a desktop, the FPGA's performance is highly competitive when considering its compact size, affordable price, and efficient power consumption.

Conclusions and Future Work

This work succeeded in adapting an existing Bird Audio Detection neural network model to a Hardware-Software implementation in a SoC-FPGA. The model was selected from a collection of algorithms related to the Bird Audio Detection Challenge, quantized and optimized, achieving an accuracy of 79.5% with an evaluation performance of 679.1ms. The model was quantized to 4 bits at the CNN section using Quantization-Aware Training, quantized to 8 bits at the RNN section using Post-Training Quantization using truncation, and the last 2 model layers remaining unquantized to floating-point. The Bidirectional GRU layers had their number of cells reduced from 64 to 1 without a major impact on accuracy, from 85.46% to 84.91% (quantized models), a 0.55% decrease in accuracy, but a big impact in decreasing resource usage by a factor of 288x. Two hardware accelerators were developed and implemented on a Xilinx Zynq UltraScale+ SoC ZU3CG, one for the Convolutions and another for the Bidirectional GRUs. The hardware resources used by the accelerators were 101 BRAMs (46%), 0 DSPs (0%), 13971 FFs (9.9%), and 14366 LUTs (20.36%).

This work explored the less-known process of quantizing a TensorFlow model and preparing it for a hardware implementation without relying on TensorFlow Lite. Quantization using QKeras and extraction of model weights from the trained model. The GRU layer was explored and successfully implemented in hardware, which is less discussed in literature when compared to Convolution layers.

This work can be improved by developing a QKeras replacement layer for the GRUv2 present in TensorFlow. QKeras has a GRU implementation called QGRU, however, it

is based on the GRUv1 TensorFlow layer. This will make it possible to quantize using Quantization-Aware Training in the RNN section of the model, replacing the quantization by truncation. The QKeras library also does not have an implementation for the Time-Distributed layer, developing one will make it possible to quantize the last layers of the model since QDense is implemented. This will make it possible to quantize the model from start to finish using Quantization-Aware Training using QKeras.

References

[1] Arjun Pankajaksha, Anshul Thakur, Daksh Thapar, Padmanabhan Rajan, and Aditya Nigam. “All-conv net for bird activity detection: Significance of learned pooling”. (2018), [Online]. Available: http://faculty.iitmandi.ac.in/~padman/papers/learnt_pooling_cameraReady_interspeech2018.pdf.

[2] Gilman RT Lewis RN Williams LJ. “The uses and implications of avian vocalizations for conservation planning”. (Aug. 2023), [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7984439/.

[3] Florence and Dan. “Birdvox”. (Aug. 2023), [Online]. Available: https://wp.nyu.edu/birdvox/.

[4] BirdVox porject. “Birdvox detailed”. (Aug. 2023), [Online]. Available: https://zenodo.org/record/1208080.

[5] Vincent Lostanlen, Justin Salamon, Andrew Farnsworth, Steve Kelling, and Juan Pablo Bello, “Birdvox-full-night: A dataset and benchmark for avian flight call detection”, in *Proc. IEEE ICASSP*, (Calgary, Canada), 2018.

[6] Panagiotis Antoniadis. “Calculate the output size of a convolutional layer”. (Aug. 2023), [Online]. Available: https://www.baeldung.com/cs/convolutional-layer-size.

[7] DCASE2018. “Bird audio detection challenge”. (Aug. 2023), [Online]. Available: https://dcase.community/challenge2018/task-bird-audio-detection.

[8] Keras. “Dense layer”. (Aug. 2023), [Online]. Available: https://keras.io/api/layers/core_layers/dense/.

[9] Kevin Ezra. "Dense layer in tensorflow". (Aug. 2023), [Online]. Available: https://iq.opengenus.org/dense-layer-in-tensorflow/.

[10] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pages 2278–2324, 1998. DOI: 10.1109/5.726791.

[11] Shulin Zeng, Kaiyuan Guo, Shaoxia Fang, Junlong Kang, Dongliang Xie, Yi Shan, Yu Wang, and Huazhong Yang. "An efficient reconfigurable framework for general purpose cnn-rnn models on fpgas". (Nov. 2018), [Online]. Available: https://nicsefc.ee.tsinghua.edu.cn/nics_file/pdf/8d00a77a-170c-4889-8bb5-9a0a148816d5.pdf.

[12] Freesound team. "Freesound". (Aug. 2023), [Online]. Available: https://freesound.org/.

[13] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using rnn encoder-decoder for statistical machine translation". (Jun. 2014), [Online]. Available: https://arxiv.org/abs/1406.1078v3.

[14] Simeon Kostadinov. "Understanding gru networks". (Aug. 2023), [Online]. Available: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

[15] Michael Phi. "Illustrated guide to lstm's and gru's: A step by step explanation". (Sep. 2018), [Online]. Available: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

[16] Anouar Nechi, Lukas Groth, Saleh Mulhem, Farhad Merchant, Rainer Buchty, and Mladen Berekovic, "Fpga-based deep learning inference accelerators: Where are we standing?", *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, 2023, ISSN: 1936-7406. DOI: 10.1145/3613963. [Online]. Available: https://doi.org/10.1145/3613963.

[17] Keras. "Maxpooling2d layer". (Aug. 2013), [Online]. Available: https://keras.io/api/layers/pooling_layers/max_pooling2d/.

[18] microfaune. "Microfaune". (Oct. 2020), [Online]. Available: https://github.com/microfaune/microfaune.

[19] microfaune. "Microfaune_ai". (Oct. 2020), [Online]. Available: https://github.com/microfaune/microfaune_ai.

[20] Google. “Qkeras: A quantization deep learning library for tensorflow keras”. (Aug. 2023), [Online]. Available: https://github.com/google/qkeras.

[21] MathWorks. “What is quantization?” (Apr. 2024), [Online]. Available: https://www.mathworks.com/discovery/quantization.html.

[22] Niklas Donges. “A guide to recurrent neural networks: Understanding rnn and lstm networks”. (Aug. 2023), [Online]. Available: https://builtin.com/data-science/recurrent-neural-networks-and-lstm.

[23] Milton Collins, Luis Garrido, Eric Jiang, and Patrick Karol. “Bird-emann”. (2018), [Online]. Available: https://www.ece.rutgers.edu/sites/default/files/capstone/capstone2018/posters/S18-09-poster.pdf.

[24] Campus Datacamp. “Dense and timedistributed layers (video)”. (Aug. 2023), [Online]. Available: https://campus.datacamp.com/courses/machine-translation-in-python/implementing-an-encoder-decoder-model-with-keras?ex=9.

[25] Leevo. “What is the interest of timedistributed after an lstm layer?” (Mar. 2020), [Online]. Available: https://datascience.stackexchange.com/a/69816.

[26] José Pedro Castro Fonseca. “Fpga implementation of a lstm neural network”. (Jul. 2016), [Online]. Available: https://repositorio-aberto.up.pt/bitstream/10216/90359/2/138867.pdf.

[27] Florence and Dan. “Warblr”. (Aug. 2023), [Online]. Available: https://www.warblr.co.uk/.