# Real-Time Bird Audio Detection using AI on FPGAs

Rodrigo Silva[a]
[a]Department of Electronical Engineering Telecommunications and Computers,
Instituto Superior de Engenharia de Lisboa, Portugal
a41429@alunos.isel.pt

***Abstract*— The evolution of digital systems to provide high-performance, low-power, and small size systems, has enabled efficient monitoring wildlife in their habitat from audio samples. The Bird Audio Detection Challenge (BADC) concerns the propose of Neural Network models to detect birds from audio samples. From the contributions to this challenge, a model was selected, trained using the provided datasets, and a hardware accelerator was developed for implementation on a Xilinx Zynq UltraScale+. The model's weights and activations are quantized and fine-tuned to improve the hardware performance, reducing resource usage while still providing valid results. The system's 79.5% accuracy is lower than the Python model's 89.75%, but this is acceptable because the goal is to reduce the model, implement it in hardware, and achieve an evaluation time of 1 second or less. The performance evaluation of the proposed system provided an estimate within 679ms. Thus, fulfilling the initial target delay of 1s.**

**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**

## I. Introduction

Birds are usually set as an example of animal behavior monitoring, such as measuring biodiversity, population surveillance, and migratory pattern analysis. However, conducting visual surveys in remote or ecologically sensitive areas such as forests can be challenging and disruptive. Sound-based wildlife detection, on the other hand, offers a non-intrusive method for gaining insights into avian behaviors and trends while minimizing disruption to natural habitats.

In this work, a SoC-FPGA system architecture is proposed to acquire and process the audio locally, sending the evaluation result to the central station, resulting in a compact system with improved energy efficiency.

This project adapts an existing Bird Audio Detection (BAD) model, implements its, and optimizes its performance using a SoC with built-in FPGA.

This paper is organized as follows: section 2 presents the background necessary to understand this work, section 3 explains the project workflow, model selection, and quantization, section 4 presents the model layers used by the model, section 5 describes the HW/SW system architecture, section 6 the accuracy and performance results of the project and section 7, the conclusions and future work.

## II. Background

### A. Bird Audio Detection

Detecting the presence of birds precedes bird counting or classification. In general, the process of Bird Audio Detection (BAD) can be split into two categories: 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, and is the one used in this work. The second method focuses on recognizing different bird species by their unique sounds.

Algorithms for this purpose have been researched and developed [5] [6] [9], driven by the Bird Audio Detection Challenge (BADC) as part of the Detection and Classification of Acoustic Scenes and Events [1] 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. The algorithms developed that entered this challenge can be found on the Bird Audio Detection Challenge Results (2018) page [2].

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.

### B. Neural Networks

The model explored in this work combines Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN).

The CNN is a type of artificial neural network designed for processing grid-like data, such as images and videos. The CNN layers the explored model uses are Convolution2D, BatchNormalization, MaxPooling2D, and Dense.

The 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. The RNN layer the explored model uses is the Gated Recurrent Unit (GRU).

### C. 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 [7] is the process of mapping floating-point values to a smaller set of discrete finite values. It is often used in various fields such as signal processing, data compression, and machine learning. In machine learning, quantization can be applied to model parameters and activations, to reduce memory usage and arithmetic complexity, while preserving accuracy to some extent.

## III. Model Design and Optimizations

### A. Project Workflow

The workflow began by finding bird audio detection models that were related to the BADC. Then, it was determined which ones were suitable for hardware implementation and validated their accuracy against the reported by the model developer.

Once the model was selected, its weights were extracted using a custom Python script. At this stage, these weights were still in floating-point format, serving to validate the replicated model in C. Additionally, the Python model was executed to evaluate some input audio files, allowing the extraction of the output of each layer to further validate the C implementation.

The subsequent step involved quantizing the model to reduce the memory footprint used by the weights and the complexity of the arithmetic operators, while attempting to maintain the original model accuracy. The quantization was done with QKeras considering a Quantization-Aware Training approach. Multiple quantizations with different data bitwidths were explored to determine the minimum amount of bits required for network operation with minimal accuracy reduction.

The quantized weights were then extracted using a custom Python script, adapted to support QKeras quantized weights. The output was 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.

The optimization of the model in High-Level Synthesis (HLS) involved 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 IPs were designed with the HLS model code and then programmed into the FPGA.

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

### B. Model Selection

From the many deep learning models found that were related to bird audio detection, the one that was considered the most relevant was the Microfaune_ai [8] model. This model did not enter the BADC but uses the same datasets for its training and evaluation.

The model receives an input of 431x1x40 features, and after the 1st Conv2D, it is transformed into 431x64x40. This model consists of 3 sections, the CNN, the RNN, and the Dense section. The output is a floating-point value from 0 to 1, and if multiplied by 100, it represents how sure the model is about the presence of a bird vocalization.

The CNN section comprises of series of Conv2D followed by BatchNormalization layers and a ReLU activation. This pattern repeats 6 times and after each odd number, it has a MaxPool2D layer reducing the number of columns of the input by half each time, from 40 to 20, 10, and 5. This section ends with a ReduceMax reducing the 5 columns down to 1, 431x64x1.

The RNN section consists of 2 Bidirectional GRU layers, both producing an output of 431x128.

The Dense section has 2 TimeDistributed Dense layers followed by a ReduceMax. The first TD Dense concentrates the input into an output of 431x64, and the second to an output of 431x1. The ReduceMax then returns a simple output, 1 value, the highest from the 431 lines.

The accuracy of this model was 90.50%, with figure 2 showing the Microfaune_ai model training progression with 100 epochs and 100 steps per epoch.

### C. Model Quantization & Weights Extraction

The model quantization was done using the QKeras [3] library. This library is a quantization extension to the Keras library, adding drop-in replacement for some of its layers, to achieve a Quantization-Aware Training.

The layers replaced were Conv2D and BatchNormalization by QConv2DBatchnorm, and ReLu by QActivation configured as ReLU. It was not possible to replace all the layers of the model with their quantized counterpart because some were not implemented by QKeras or the implementation did not match the one used by the model. The Dense layer could be replaced by QDense, but since a replacement for the Time-Distributed wrapper does not exist, it was not possible. The Bidirectional wrapper (QBidirectional), and GRU layer (QGRU) exist but were not used because the implementation of the GRU layer in QKeras is based on the GRUv1 of TensorFlow. The Microfaune_ai model uses the GRUv2 of TensorFlow.

After exploring multiple configurations of quantization, the chosen one was quantizing the model with 4-bit with 1 integer, Q(4,1), for the CNN section of the model. The RNN section of the model containing the Bidirectional GRUs was quantized by truncation to 8-bit and 1 integer, Q(8,1). The Dense section was not quantized because of the small layer size when compared to the rest of the model, and to reduce further impact on the model accuracy. This final section is executed on the ARM processor using floating-point values instead.

During the quantization exploration, it was noticed that reducing the number of GRU cells did not have a major impact on the model, with the only noticeable impact being a slower training progression. There was a major positive impact on the reduction of weights, reducing them from a total of 124416 to 432 weights, a reduction by a factor of 288x. The accuracy of the quantized model with 64 GRU cells was 85.46% while using only 1 GRU cell was 84.91%.

Figure 3 shows the quantized Microfaune_ai model training progression with 100 epochs, 100 steps per epoch, and 1 GRU cell.

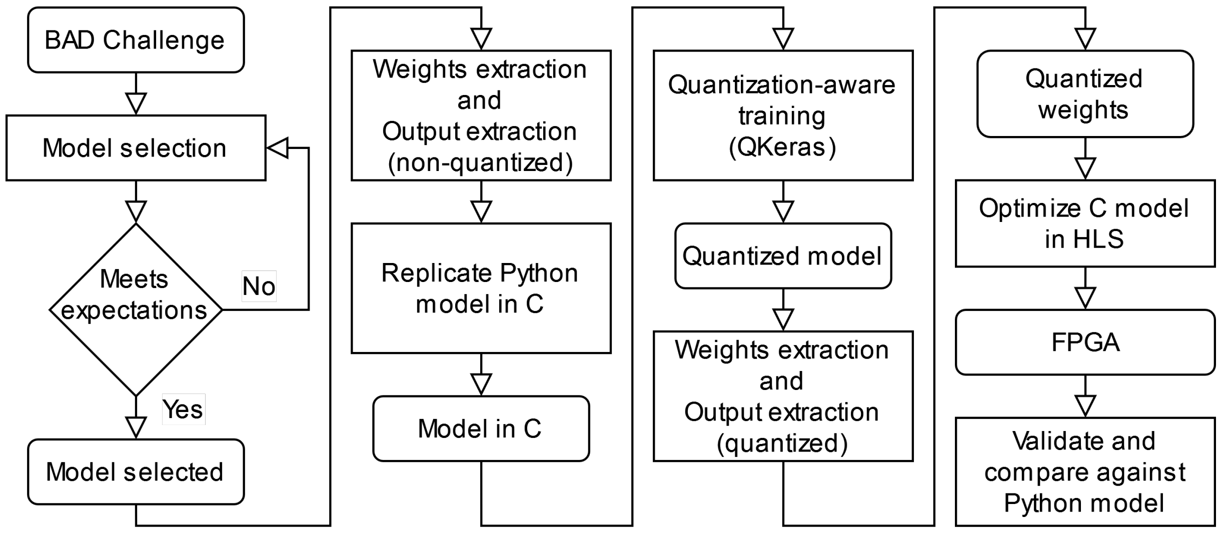
BAD Challenge
Model selection
Meets expectations
No
Yes
Model selected
Weights extraction and Output extraction (non-quantized)
Replicate Python model in C
Model in C
Quantization-aware training (QKeras)
Quantized model
Weights extraction and Output extraction (quantized)
Quantized weights
Optimize C model in HLS
FPGA
Validate and compare against Python model

Fig. 1. Project workflow, from model selection to FPGA deployment.

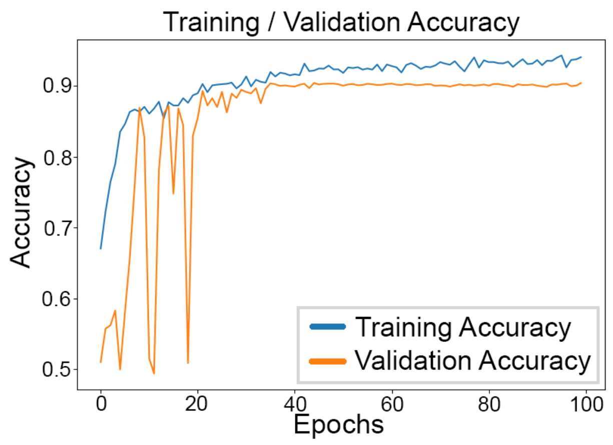

Fig. 2. Original microfaune_ai, accuracy training progression plot, 90.50%.

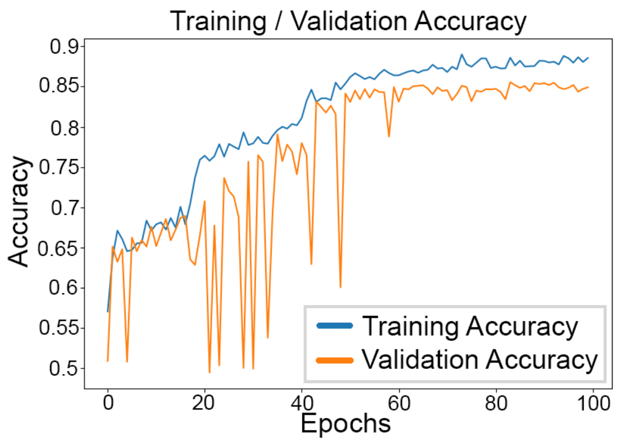

Fig. 3. CNN quantized to 4 bits and RNN 1 cell, accuracy training progression plot, 84.91%.

After model quantization, the weights were extracted using a custom Python script for later use in the hardware implementation. To achieve this, a custom Python script was developed that loads the quantized model and then iterates through its multiple layers and then stores all the weights sequentially in a binary file. QKeras layers also have a scale associated with their weights, which were merged before storing the associated weight.

## IV. Embedded Software Microfaune Model

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 simple test environment in Python consisted of creating smaller models to debug and understand how each layer worked. These small models only consisted of the layer being studied, and executing them using small input sizes.

In the second step, some implementations were found in the TensorFlow Python implementation, especially for the GRU implementation which proved to be difficult to find online. After this, the TensorFlow implementation was stripped out of TensorFlow and implemented using the Numpy Python library, a necessary step to further understand each layer before replicating them in C.

Replicating the model in C was done layer by layer, adding each layer by layer and comparing the output at each stage against the Python implementation.

### A. *Conv2D & BatchNormalization*

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. After each Conv2D, there is a BacthNormalization layer which was merged with the Conv2D weights during quantization in QKeras, using the QConv2DBatchnorm layer, making it possible to remove the BatchNormalization layer for evaluation. The Conv2D layer was replicated to process the input by filter, line, and column. The first Conv2D receives an input with 1 channel, while the rest 64 channels.

### B. *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.

### C. *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.

### D. *Bidirectional GRU*

The Bidirectional layer provides the wrapped layer with the same input in a forward and backward direction. The forward direction gives the wrapped layer the input exactly like the one received from the previous layer. The backward direction gives the wrapped layer the input with a flipped order of lines, from end to beginning, without changing the order of the columns or channels.

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 and in the Keras GitHub repository [4]. This layer implementation was studied using the Keras print tensor function that would print to the console the values at each calculation of the GRU step function during an evaluation. Both Bidirectional GRUs produce an output of 431x128, which consists of the output of both directions merged, 431x64 each.

### E. *Time-Distributed Dense*

The Time Distributed layer is a wrapper that allows the application of a specific layer, in this case, the Dense layer to each time step. The number of time steps used is based on the number of units the Time-Distributed is configured to.

The 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.

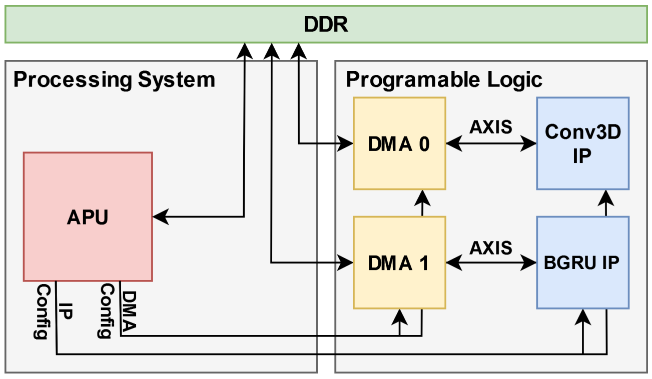

Fig. 4. Proposed system architecture for Bird Audio Detection.

## V. HW/SW System for Bird Audio Detection

The proposed system architecture, figure 4, 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 the CPU using floating-point values. The last Reduce Max layer of the model only processes a small 431 by 1 vector, so it is also processed on the 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 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.

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 DMA blocks and the IP blocks.

### A. *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.

This block is divided into 5 sub-blocks: Input Buffer, Conv3D, MaxPool2D, Output Buffer, and Weights. It receives the Input over AXI-Stream and the configuration, the index of the layer to be processed ranging from 0 to 5, over AXI-Lite.

The Input Buffer sub-block stores data from the input stream. It has enough storage for three lines of the input map, with each value being 4 bits wide.

The Conv3D sub-block is responsible for processing the input according to the Convolution algorithm, and sending it to the next sub-block. The next sub-block is determined by the configuration index, which holds information if MaxPool2D should be processed or if the output should be sent to the Output Buffer sub-block.

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, 111552 bytes total.

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, and 431x64x10. To decrease development complexity and time, this Conv3D IP only processes inputs with 64 channels. This means that the Input and Kernel Weights must be padded with zeros for the first Convolution. The Input originally is 431x1x40 and must be padded to 431x64x40 with zeros. As for the Kernel, originally is 3x3x1x64 and must be padded to 3x3x64x64 with zeros.

## B. 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, using 1 GRU Cell.

This block is divided into 4 sub-blocks: Input Buffer, Bidirectional GRU, GRU Cell, and Weights. It receives the Input over AXI-Stream and the configuration, the index of the layer to be processed ranging from 0 to 3, over AXI-Lite. The configuration index is 0 to 3 instead of 0 to 1, because the even indexes are related to the forward direction, and the odd to the backward direction.

The Input Buffer sub-block stores the entire feature map, each value being 8 bits wide, returned by the previous layer: 431x64 on the first Bidirectional GRU layer and 431x2 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, 432 bytes total.

## C. 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. The different sections of this Optimized model use different data widths. The CNN uses 4 bits, the RNN 8 bits, and the TD Dense floating-point values. This post-processing is done by the CPU of the Processing System.

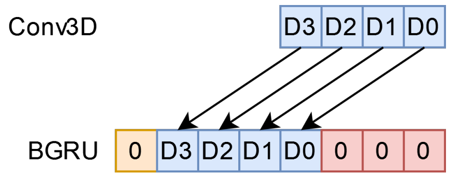

Fig. 5. Output conversion from Conv3D to BGRU, 4 bits to 8 bits.

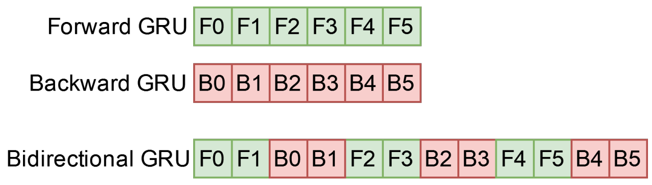

Fig. 6. Example of BGRU post-process for GRUs with 2 cells.

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, as shown in figure 5. 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).

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. 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, as shown in the example figure 6. Since this example 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.

Similar to the conversion between Conv3D and BGRU, RNN to Time-Distributed requires a conversion from 8 bits to floating-point. For this purpose, it was developed a function that receives an 8-bit fixed-point value and returns the respective floating-point value.

## D. Hardware Design

The Conv3D and BGRU IPs were designed using High-Level Synthesis (Vitis HLS) and integrated with the processor using Vivado. 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.

## E. Software Design

The Microfaune_ai model was replicated with software that uses the FPGA implemented layers.

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

Then a Model Predict is called that 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, processing the CNN section of the model. 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.

The CNN to RNN transition comes, in which the CPU is used to transform the 4 bits into 8 bits, creating the RNN input.

In the RNN section, 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. The BGRU IP is called 2 more times, executing the same job again, but this time using the input from the previous call.

The RNN to TD Dense transition comes, in which the CPU is used to transform the 8 bits into floating-point values, creating the TD Dense section input.

The model execution ends after the next layers complete their execution, returning a single output float value called Global Score.

## VI. Accuracy & Performance Results

With the model implemented in hardware, a series of tests were conducted, comparing the Original model, the Modified that uses only 1 GRU cell, the QKeras Modified model, and the Optimized on a FPGA based on the Modified model. These tests all used the same 400 samples, 200 with positive features and 200 with negative features. The performance results were terminated by averaging the execution time across the 400 samples, while the resource usage metrics were obtained from Vivado's FPGA resource usage report. The chosen SoC-FPGA was the Xilinx Zynq UltraScale+ ZU3CG SoC.

### A. *Model Accuracy*

For this sample size, the Original model reported an accuracy of 89.75%, the Modified an accuracy of 86.25%, the QKeras Modified 80%, and the Optimized hardware implementation 79.50%. The Optimized had a reduction in accuracy of 1.5% when compared to its baseline, the QKeras Modified, a model that is based on the Modified but uses the QKeras quantization as described earlier.

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

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 II. 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.

TABLE I
Accuracy between the Original, Modified, and Optimized models.

| | Count | Percentage |
|---|---|---|
| **Original 64 cells** | 359 | 89.75% |
| **Modified 1 cell** | 345 | 86.25% |
| **QKeras Modified** | 320 | 80.00% |
| **Optimized** | 318 | 79.50% |

TABLE II
Weights memory usage comparison between a model with 64 GRU cells and 1 GRU cell

| | Model 64 cells | Model 1 cell | Red. Fact. |
|---|---|---|---|
| **Convolutions** | 111552 B | 111552 B | 1 |
| **1st GRU** | 49920 B | 402 B | 124.2 |
| **2nd GRU** | 74496 B | 30 B | 2483.2 |
| **Non-Quantized** | 66052 B | 66052 B | 1 |
| **Total** | 302020 B | 178036 B | 1.7 |

### B. *Performance and Resources*

The Optimized implementation on the FPGA was compared against a software implementation running on the ARM CPU of the Ultra96-v2. This software implementation is based on the Optimized hardware implementation, but instead of targeting HLS, it was implemented in C by replacing AXI-Lite and AXI-Stream interactions with C pointers for memory transactions.

Table III shows the execution time improvement, as well as the speed-up factor achieved by the Optimized hardware implementation. The software implementation of the Optimized model takes around 53 seconds for each input evaluation, while the Optimized model that uses the FPGA for the Convolutions and GRUs takes around 679 milliseconds.

This massive reduction in execution time will increase the chances of identifying a bird vocalization when deployed in the field. Since the input is an audio recording of 10 seconds, the software implementation can only evaluate one input per minute, with an audio recording uptime of 10.8%. When using the Optimized hardware implementation it can do 6 input evaluations per minute, with an audio recording uptime of 95.7%.

Table IV shows the FPGA resource usage. Starting with

TABLE III
Comparison of milliseconds used in different sections of the model, between ARM CPU and Optimized.

| | CNN | RNN | CPU Layers | Total |
|---|---|---|---|---|
| **ARM** | 53477.6 | 10.52 | 7.423 | 53495.3 |
| **Optimized** | 666.4 | 4.646 | 7.431 | 679.1 |
| **Speed Up Factor** | 8025% | 226% | 99% | 7877% |

TABLE IV
FPGA RESOURCE USAGE TAKEN BY THE IPS AND OPTIMIZED IMPLEMENTATION.

| | **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% |

the Conv3D IP that processes the Convolutions, the BGRU IP that processes the Bidirectional GRU layers, the Optimized showing the total resource usage for the hardware implementation, and the total hardware resources available in the ZU3CG FPGA. The usage percentage is the relation between Optimized and the ZU3GC FPGA.

## VII. CONCLUSIONS

This work succeeded in creating a neural network model from Bird Audio Detection in 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.

## REFERENCES

[1] DCASE2018. ”Bird Audio Detection Challenge”. Available: https://dcase.community/challenge2018/task-bird-audio-detection, July 2018. [Online].

[2] DCASE2018. ”Bird Audio Detection Challenge - Results”. Available: https://dcase.community/challenge2018/task-bird-audio-detection-results, July 2018. [Online].

[3] Google. ”QKeras: a quantization deep learning library for Tensorflow Keras”. Available: https://github.com/google/qkeras, May 2024. [Online].

[4] Keras. ”GRUv2 - standard_gru.step”. Available: https://github.com/keras-team/keras/blob/94a1712027366cd626df4f1d77f25f918a18f456/keras/layers/rnn/gru.py#L9691, May 2024. [Online].

[5] Mario Lasseck. Acoustic bird detection with deep convolutional neural networks. Technical report, DCASE2018 Challenge, September 2018.

[6] Sidrah Liaqat, Narjes Bozorg, Neenu Jose, Patrick Conrey, Antony Tamasi, and Michael T. Johnson. Domain tuning methods for bird audio detection. Technical report, DCASE2018 Challenge, September 2018.

[7] MathWorks. ”What Is Quantization?”. Available: https://www.mathworks.com/discovery/quantization.html, Apr 2024. [Online].

[8] microfaune. ”microfaune_ai (updated fork)”. Available: https://github.com/W-Alphonse/microfaune_ai, Oct 2020. [Online].

[9] Fabio Vesperini, Leonardo Gabrielli, Emanuele Principi, and Stefano Squartini. A capsule neural networks based approach for bird audio detection. Technical report, DCASE2018 Challenge, September 2018.