AI Voice Processing on an FPGA: Final Report

Andrew Rooney
ECE 492
University of Alberta
Edmonton, CAN
arrooney@ualberta.ca

Nathan Mikhail
ECE 492
University of Alberta
Edmonton, CAN
mikhail@ualberta.ca

Justin Diala
ECE 492
University of Alberta
Edmonton, CAN
justineb@ualberta.ca

Ramana Vasanthan
ECE 492
University of Alberta
Edmonton, CAN
ramana@ualberta.ca

Abstract*—We sought to demonstrate the feasibility of using machine learning (ML) to accelerate audio noise suppression and keyword detection tasks on an edge device using synthesized hardware logic on a field programmable gate array (FPGA). ML has the potential to outperform the current state of the art audio filtering systems, especially at tasks such as filtering babble noise, where the noise exists in the same spectral range as the signal. However, many ML algorithms are computationally expensive and require cloud-based resources to do inference. A system prototype has successfully been designed, developed, and tested, and in this paper, we present our prototype system which uses a Digilent Genesys ZU-3EG development board to accelerate a recursive neural network model to suppress stationary, non-stationary, and babble noise. A keyword detection model was trained using Google's "teachable machine" which can run without cloud resources but requires a graphical processing unit. The proposed ML filtering system imposes a latency of 27ms and successfully suppresses noise although with minor auditory artifacts of the software processing. The filter improves the signal to noise ratio (SNR) of white (stationary) noise and babble noise by 45dB and 30dB respectively; the SNR of non-stationary noise was not improved by the system. Keyword detection is done by sending voice input into a TensorFlow.js model that is trained to detect certain keywords. If certain keywords have been detected, then a user interface (UI) gives user feedback that the word has been detected. The latency of this system was measured to be about 50ms, with an accuracy of 0.978.

***Keywords*—Machine learning, noise suppression, recurrent neural network, real time, high-level synthesis, system on a chip**

I. Introduction

Machine learning (ML) models can solve a vast array of data processing problems efficiently, and without the need for highly specialized algorithms. Audio processing is no exception; however, most ML based audio filtering solutions such as Discord's "Krisp" or Audo AI require cloud-based compute resources, and thus require a good internet connection, precluding them from use on edge devices with limited or no network connectivity and limited compute resources.

This project was motivated by a desire for an adaptive, ML based audio filtering solution that is capable of filtering stationary and non-stationary noise across the frequency spectrum without the need for cloud-based, or desktop-level resources. This includes filtering audio in the same frequency range as the human voice (which is difficult for non-ML approaches), from a speaker's environment while remaining compact enough for deployment in military vehicles or the helmet of a soldier or firefighter. Keyword detection could also be used in military scenarios to operate controls hands-free.

By using an ML approach to audio filtering, models could be trained to excel at filtering sound that is expected in the specific deployment scenarios and could be trained to work well with a specific individual's voice. Moreover, a well selected and trained ML audio filter could outperform the state-of-the-art adaptive filtering solutions, such as the Wiener filter, especially in situations where the noise overlaps with the frequency range of human speech [1].

The system that has been developed for this project serves as a proof of concept of the feasibility of selecting or creating an ML model that (a) is highly capable of suppressing noise in an audio signal, (b) can be synthesized on programmable logic (PL), and (c) can run in real time. This system is indicative of the rise of ML as a convenient and powerful solution for handling data. Using FPGA resources to enable real time ML inference could have far reaching implications beyond noise suppression.

In this paper we present and discuss a ML based prototype solution we created to achieve the desired audio processing as mentioned above. We will outline our created designs for hardware, software, and firmware components, as well as a separate keyword detection system. Design decisions will be outlined and comparisons to functional and performance requirements will be made. Lastly, we will explain the results we achieved for these systems and conclude with a discussion of potential future work on the project.

II. Prototype Design, Implementation and Results

The first step in defining the goals and scope of this project was the project proposal, and proposal response. In the proposal response [2] we defined our goals with our client: to develop a system that could suppress stationary and non-stationary noise in real time using a machine learning (ML) model. The system was to make use of a field programmable gate array (FPGA) to accelerate ML inference and respond to keyword commands. This project is primarily research and development (R&D) focused, and the prototype presented here serves as an initial proof of concept for deploying a real time

ML based audio filtering on resource constrained edge devices. As such, the prototype form factor and landed production cost are not key aspects of the design because the proposed designs are not intended to be manufactured at a large scale or be deployed to end users.

Key aspects of the target prototype are captured in the requirements that are enumerated in the project preliminary design document (PDD) and PDD update documents [3], [4]. The key functional requirements state that the system should be able to: (FR-1) filter stationary and non-stationary noise from input, (FR-2) accept human voice as input, (FR-3) output filtered audio, (FR-4) operate on an edge platform, and (FR-5) respond to keywords [4]. The performance requirements dictate that the system must: (PR-1) operate in real time, (PR-2) sample audio at at least 48KHz, (PR-3) fit on an embedded edge platform, (PR-4) have latency of less than ¼ second, and (PR-5) respond to keywords with specified behavior [4]. Our prototype's compliance with these requirements will be discussed throughout this section.

The system architecture proposed in [4] illustrates the deployment architecture and components of our system; the key functional nodes are the audio input/output, audio pre-/post-processing, the audio filtering ML model, and keyword detection and action. These high-level nodes became the main development tasks of our system which are reflected in our project's Gantt chart [5]. In the Gantt chart we planned to have audio input/output capabilities developed by February 18, the audio pre-/post-processing was to be developed by March 7th. The audio filtering ML model research and hardware implementation was to be done by March 22 along with the system software and keyword detection models. The work breakdown and timeline is shown in much greater detail in [5]. Overall, despite updates to the design throughout the project, we were able to adhere to our target timeline.

A closer look at our prototype and the relevant component selection will be broken down into the components for the audio filter, and keyword detection separately because these system's implementations in our prototype are entirely separate.

A. Audio Filtering

An overview of the hardware, software, and machine learning components of the audio filtering system prototype will be presented along with the key results and requirements analysis.

1) Hardware

The hardware for our prototype is relatively minimal, and we had no major issues or difficulties with our hardware. The central hardware component of our audio filter design is the Digilent Genesys ZU-3EG development board [6], which houses the Xilinx Ultrascale+ MPSoC (multi processor system on a chip) [7]. This MPSoC was chosen for its large FPGA fabric, which offers 71,000 look-up tables (LUTs) to support an ML model and has 4 Cortex A53 central processing unit (CPU) cores and 2 Cortex R5 CPU cores; all of these components share access to double data rate (DDR) random access memory (RAM). The MPSoC functions to support the real time, and edge device requirements (FR-4, PR-1) of the audio filtering system.

A peripheral module interface (PMOD) I2S2 audio I/O device [8] is used to connect line-in and line-out audio devices to the Genesys development board; this component supports audio sampling rates up to 200KHz, and thus supports the requirements PR-2, FR-2, and FR-3. Any line-in and line-out audio devices can be used with our prototype system.

A more rigorous discussion of the hardware components and their connectivity for our prototype can be found in [4].

2) Firmware and Software

Firmware in our system refers to the intellectual property (IP) cores that are used in the FPGA to execute low-level actions. The firmware in our prototype design has two main responsibilities: controlling the audio codec and delivering audio data to and from the CPU processing system (PS) and executing the neural network inference on audio data.

An open-source IP from Digilent [9] is used to control the PMOD codec, and Xilinx's direct memory access (DMA) IP [10] is used to send and receive audio data from the DDR RAM that is shared with the PS.

The neural network is also connected to the PS and is implemented by a custom IP which was created through the use of high-level synthesis (HLS). Initially, we explored the use of a convolutional neural network (CNN). However, using similar tools to synthesize the model from high level representations into register transfer level (RTL) code, we determined the CNN would be far too resource hungry for our available devices. Specifically, only 1 of 16 CNN layers was requiring >1 million LUTs, over 1400% of what was available.

We then shifted our focus to Mozilla's RNNoise [11] was used as a pre-trained C-based model, which we used as the starting point for hardware implementation. The RNN's recurrent connections give it the ability to model time sequences; this makes it an ideal architecture for audio processing. After making architectural modifications, the model was able to be synthesized but used floating point arithmetic. This resulted in 213% LUT utilization on our targeted ZU-3EG MPSoC. Additionally, the floating-point model had an end-to-end latency of 3.8ms at 300MHz. This was improved by using (22,14) fixed-point arithmetic, and along with other optimizations yielded only 67% LUT utilization and 0.25ms of latency per 10ms of audio data, which supports real-time operation and thus PR-4. Table I displays these optimizations and more.

As seen in Table I, implementing fixed-point arithmetic resulted in a significant decrease in resource utilization as well as overall latency. This allows us to operate our neural network at lower power, and on smaller edge devices fulfilling requirements PR-3 and FR-4.

TABLE I. HLS REPORTS FOR FIXED AND FLOATING POINT SOLUTIONS

	Look-Up Tables [%]	**Flip-Flops [%]**	**Digital Signal Processors [%]**	**Latency [ms]**
Floating-Point	213	275	130	3.8
Fixed-Point	67	31	98	0.25
Percent Reduction	**69**	**89**	**25**	**93**

The Xilinx Zynq IP [7] is used to interface these firmware components in the FPGA with the rest of the MPSoC system.

Software runs on the Cortex A53 CPU; its purpose is to initiate and respond to DMA transfers of audio data to and from the FPGA, perform the audio pre- and post-processing, and execute the ML filtering inference of 10ms samples of audio data on the FPGA. The FreeRTOS operating system [12] is used to perform these duties in 5 separate tasks. 2 of the tasks are asynchronous (event based), and respond to DMA slave to master, and master to slave interrupts. 2 tasks buffer data at the input and output of the audio filtering system, and the fifth task does all the audio processing (pre-/post-processing and ML inference); all of the tasks communicate through queues. This design uses the tasks as a data pipeline to provide elastic data storage at the input and output of the filter and is intended to provide the smoothest audio possible.

It initially proved difficult to get this software to run fast enough to support our real time requirements. Given that the ML model (including the pre- and post-processing) operates on 10ms buffers, we had to be able to do the processing in less than 10ms to support real-time throughput. At first the software took ~100ms to process 10ms of audio data - this figure was dominated by the preprocessing. The preprocessing consists of applying a Vorbis window, performing a Fourier transform, and separating the spectral information into quantized regions based on the bark scale [11]; this processing uses floating points and is expensive software. However, we were able to increase the speed to just over 10ms by letting the compiler optimize for fast floating-point math and optimizing the instructions for our particular CPU and architecture. The final compiler flags used were: `*-O3 -ffast-math -mtune=cortex-a53 -march=armv8.1-a -mcpu=cortex-a53*`. We got another speed decrease to 0.7ms by reconfiguring the software to flush the DDR RAM CPU cache at the memory lines where DMA transfers take place, as opposed to globally disabling the CPU cache. We also found through experimenting that running the real time operating system (RTOS) scheduler at 900Hz, and buffering 20ms of audio data at the input of the filter yielded the smoothest sound (a subjective measurement). The latency of each processing step was measured with the Xilinx timer application programmer's interface (API), and the total audio latency - which is dominated by buffering at the input and output - was also measured. The results are shown in Table II.

From the measurements in Table II, we see that the total software processing time is 0.7ms per 10ms of audio data. This means that the system is capable of real time operation - thus fulfilling requirement PR-1. The 0.3ms for RNN inference largely agrees with the 0.25ms reported by the hardware simulations. The overall latency imposed by this filter is 27ms when software buffering is included - the software buffers 20ms of audio data before it is dispatched for processing and is also accumulated at the output. This total latency supports PR-4 within some tolerance (a 7.7% difference from the target value of 25ms). The final source code can be found at [13].

TABLE II. SYSTEM LATENCY

	System latency [ms]
Pre-processing	0.3 (/10ms sample)
Post-processing	0.1 (/10ms sample)
RNN Inference	0.3 (/10ms sample)
Total latency (including software buffering)	27

3) Filter Results

Performance of the filter was characterized using the C-simulation of the synthesized RNN design, which includes the fixed-point conversions. By running an audio sample with noise added to a man's voice, and by estimating the noise of the filtered audio using a quiet part of the sample, we found signal to noise ratio (SNR) measurements of several noise types before and after applying the filter. The results for white noise, babble noise (i.e., people speaking in the background), and non-stationary noise (dog barking) are best illustrated in the time domain as shown in Figures 1-3.

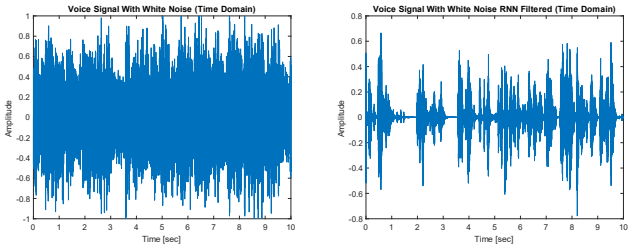

Fig. 1. Audio with stationary noise before (left) and after (right) filter

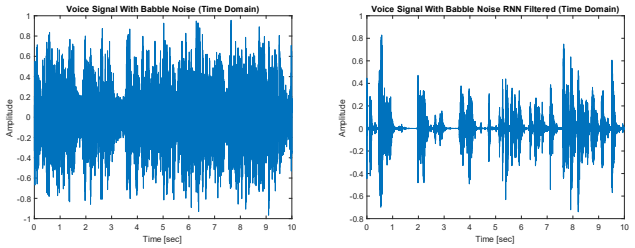

Fig. 2. Audio with babble noise before (left) and after (right) filter

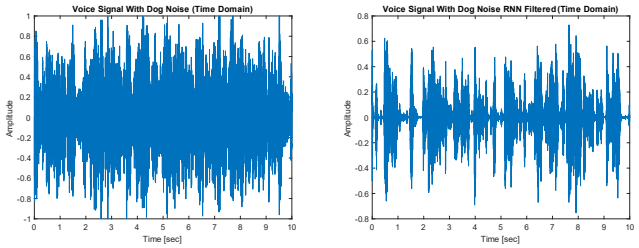

Fig. 3. Audio with non-stationary noise before (left) and after (right) filter

The SNR measurements as summarized in Table III showed an improvement of 45dB for white noise, 30dB improvement for babble noise, and no significant change for non-stationary noise. In the non-stationary case, the barking dog made it through the filter whenever the speaker was talking. These results are clearly heard in the real time operation of the device.

TABLE III. *FILTER SNR RESULTS*

	White Noise (stationary noise)	**Dog Barking (non-stationary noise)**	**Babble noise**
Unfiltered SNR [dB]	3.5004	4.4527	6.5315
Filtered SNR [dB]	48.4019	4.0554	36.7023

Overall, the filter performance was very good, and the latency was imperceptible. However, in addition to the poor performance with non-stationary noise, there was some periodic clicking that is hypothesized to be an artifact of the software processing. Refer to section III of this document for a discussion on the plans to improve the system.

B. Keyword Detection

An overview of the keyword detection system will be presented in this section, along with some results gathered from benchmarks. Design challenges and solutions will also be discussed.

1) Teachable Machine

For the keyword detection, a model was created through Teachable Machine. Teachable Machine grabs samples of the background noise and audio samples of the keyword that is to be detected. From that Teachable Machine trains a machine learning model and outputs a tensorflow.js model file.

2) Component Selection

We selected a computer that has a decently powerful GPU to leverage the WebGL capabilities of tensorflow.js so that we can meet real time requirements. We also used ubuntu so that we can quickly deploy this application for the purposes of the prototype.

3) Prototype Design

The current implementation of the keyword detection in the system was done through a web implementation of tensorflow.js that displays detection output to the browser. Currently the entire keyword detection system runs in through a computer and is composed of 2 parts: a server and a JavaScript application that has a simple UI displayed in the browser. The server hosts all the files needed for the keyword detection to run i.e., Tensorflow.js libraries and Teachable Machine model. A simple UI was run that grabs audio from the FPGA via line out and inputs that into the machine learning model. The machine learning model will then display an output to the UI based on the word detected.

4) Results

Through our benchmarking of the model using simulated TensorFlow loads (i.e., varying tensor input size) we measured an average best time of 50ms for the model to process a tensor input and output a result, this time is an average over 1000 runs (i.e., 1000 inputs to the model). Furthermore, per epoch we gathered an accuracy in the model of 0.978 in detecting the trained keywords after running the training for 150 epochs. With these results, we can respond to keywords and perform a specified action; thus, we are compliant with FR-5 and PR-5 requirements. The system operates in real time and meets our latency requirements thus the system satisfies FR-6 and PR-1.

5) Issues

The current design of using a computer to run the keyword detection is not ideal. We initially had the idea to run the keyword detection through Petalinux but that idea was abandoned because Petalinux did not support JavaScript therefore we could not use the models from Tensorflow.js models that Teachable Machine outputs. We tried implementing the keyword detection through a raspberry pi board but failed our real time constraints due to the lack of WebGL support and the lack of a powerful GPU.

The current implementation, while effective, has drawbacks in that it relies on a powerful GPU to run in real time and has overhead in that it needs an operating system and browser to run. Thus, we are only partially compliant with the edge device requirement FR-4, given that this implementation requires resources that are beyond many edge devices.

III. FUTURE WORK

As discussed, the system presented here is meant as a proof-of-concept exploration of the feasibility of using ML to perform noise suppression on an FPGA. Further design iterations would be required to refine the system before it could be considered for deployment. Improvements would be made to the system performance in the target scenarios (i.e., the user

stories discussed in [4]). Specific areas of work to improve this system will be identified in this section for each subsystem.

A. RNN Implementation

Future work related to the voice filtering model includes additional resource optimization and further model training. A more detailed analysis and exploration of HLS capabilities and design directives may lead to further resource optimization. This would allow for more free space in the FPGA, leading to the addition of other features, or a larger and more powerful neural network. Additionally, the model could be trained more rigorously, or for more specific use cases. As mentioned, the model used was pre-trained and was likely targeted for general use. The dataset used to train the model available for download [14] suggested >6.4GB of training data was used. While general training may be beneficial for the common case, for our specific use cases in military or firefighting applications, user or environment specific training would likely be advantageous.

Additional performance requirements could include an increase in SNR for non-stationary noise, and a reduction in DSP utilization. An increase in non-stationary SNR would make the model more applicable for general purpose, or even in situations where non-stationary noise is most common. A reduction in DSP utilization would again create free space for additional features to be implemented, taking advantage of the high-speed processing a DSP can provide.

B. Software

Our system would be made simpler, and more performant by removing the software altogether. In future designs we would opt to do the audio pre- and post-processing with custom FPGA logic instead of in software; we expect that this would help to smooth out the sound (i.e., reduce the clicking that we noticed from the software processing), and would reduce the overall system complexity.

C. Keyword Detection

To further improve the keyword detection model and integrate it with the voice filtering aspect of the system, Petalinux could be used with Python on the ARM processor of the MPSoC to run the model. Python can natively be run on Petalinux by changing basic configuration settings and Python scripts were ran as a proof of concept with small third-party libraries so the next step would be to use TensorFlow on Petalinux to run the keyword detection model. The model created by Teachable Machine can be converted to a Keras model which enables us to move away from the JavaScript implementation.

A further improvement could be to implement the model as custom logic using the hls4ml library to convert the Teachable Machine model to synthesizable C code which can be input to Xilinx's HLS tools. This would eliminate the overhead from the Petalinux operating system and help with latency.

Overall, the goal of these improvements would be to have an easier interface with the voice filtering system so that its output can be multiplexed into the keyword detection model. From there, the keywords could be used to control the system as outlined in FR-5 such as by toggling filtering and mute functionality.

IV. Conclusions

Our project was created to research ML based audio processing and keyword detection systems on edge devices. Existing ML algorithms that are capable of filtering stationary, non-stationary and babble noise are powerful but require high computational resources and network interfaces, that of which is not available on edge devices. We designed, created and tested a recursive neural network-based voice filtering system, accelerated on an FPGA. Software was used to do the required pre- and post-processing; latency challenges with the software were overcome by exploiting compiler optimizations and ensuring the DRAM cache was enabled wherever possible. Additionally, we created a keyword detection prototype, showing proof of concept for such a task on an offline machine. The filtering system yielded 27ms of latency and hence produced real-time output. Additionally, it could reduce SNR of stationary noise by 45dB and babble noise by 30dB. However, it did not perform as well with non-stationary noise, resulting in no change of SNR. Regardless, the voice filtering system was successful. Future work for voice filtering includes new or additional training of the RNN and further algorithm or design improvements for a reduction in resource utilization. The voice filtering software could also be improved to be made simpler and be more performance oriented by offloading portions of the pre- and post-processing onto the FPGA. Lastly, we successfully demonstrated how keyword detection can operate on an offline device. There were challenges in implementing the model on Petalinux as initially planned because JavaScript is not natively supported on it. Thus, future work for this component could include either implementing the model on Petalinux with Python or to implement the model as custom logic using high level synthesis tools for easier integration with the voice filtering system. This way, keywords can be used to control the rest of the system for a hands-free experience.

V. Acknowledgements

We would like to thank Dr. Bruce Cockburn for lending his Digilent Genesys ZU-3EG development board to our team for this project. This addition helped make our project a success. We would also like to thank Dr. Steven Knudsen for his valuable suggestions and feedback. Lastly, we would like to thank Raju Machupalli and Stephen Fan for their support throughout the lab sessions.

VI. References

[1] P. Guduguntla, "Background noise removal: Traditional vs AI algorithms," Medium, 18-Mar-2021. [Online]. [Accessed: 29-Mar-2022].

[2] A. Rooney, N. Mikhail, J. Diala, and R. Vasanthan, "Proposal Response: AI Voice Processing Algorithm on FPGA," Jan. 2022.

[3] A. Rooney, N. Mikhail, J. Diala, and R. Vasanthan, "Preliminary Design: AI Voice Processing on FPGA," Feb. 2022.

[4] A. Rooney, N. Mikhail, J. Diala, and R. Vasanthan, "Preliminary Design: AI Voice Processing on FPGA," Mar. 2022.

[5] A. Rooney, N. Mikhail, J. Diala, and R. Vasanthan, "ECE492 AIVP Project Plan V01," Feb. 2022.

[6] “Genesys ZU Reference Manual,” Genesys ZU Reference Manual - Digilent Reference. [Online]. Available: https://digilent.com/reference/programmable-logic/genesys-zu/reference-manual. [Accessed: 05-Apr-2022].

[7] “Zynq UltraScale+ MPSoC Processing System v3.3 LogiCORE IP Product Guide.” Xilinx, 10-Jun-2020.

[8] “PMOD I2S2 Reference Manual,” Pmod I2S2 Reference Manual - Digilent Reference. [Online]. Available: https://digilent.com/reference/pmod/pmodi2s2/reference-manual?redirect=1. [Accessed: 05-Apr-2022].

[9] Digilent (2019) Pmod-I2S2 [Source code]. (Version v2018-2.1)

[10] “AXI DMA v7.1 LogiCORE IP Product Guide.” Xilinx, 14-Jun-2019

[11] J.-M. Valin, “Xiph.org / RNNOISE,” GitLab. [Online]. Available: https://gitlab.xiph.org/xiph/rnnoise. [Accessed: 16-Mar-2022].

[12] Amazon Web Services, The FreeRTOS™ Reference Manual. Amazon, 2017.

[13] FilteraFPGA, “FilteraFPGA/audio_processing_software [source],” GitHub. [Online]. Available: https://github.com/FilteraFPGA/Audio_Processing_Software. [Accessed: 05-Apr-2022].

[14] J.-M. Valin, “RNNoise: Leaning Noise Suppression,” RNNOISE: Learning noise suppression, 27-Sep-2017. [Online]. Available: https://jmvalin.ca/demo/rnnoise/. [Accessed: 07-Apr-2022].