System on Chip Architecture PYNQ-Z2 voice commands detection Project Report

Diamante Simone Crescenzo s295783, Davide Fogliato s303497, Alfredo Paolino s295152, Alessio Salta s296191
Politecnico di Torino

Contents

I Introduction 1

II Background 1
- II-A Bluetooth Low Energy (BLE) . . . 1
- II-B Machine learning & voice recognition . . . 1
- II-C FPGA accelaration . . . 2
- II-D FINN framework . . . 2

III Hardware components 2
- III-A Wireless Sensor Node (WSN) . . . 2
- III-B Central Processing Node (CPN) . . . 2
- III-C Bluetooth adapter . . . 2

IV Preliminary setup 3
- IV-A PYNQ environment setup . . . 3
  - IV-A1 VM and build environment setup . . . 3
  - IV-A2 PYNQ-Z2 image build . . . 3
- IV-B PYNQ-Z2 image flash . . . 3
- IV-C PYNQ-Z2 Bluetooth setup . . . 3
- IV-D BlueTile setup . . . 4
  - IV-D1 Build process for BlueTile . . . 4
  - IV-D2 Flash process for BlueTile . . . 4
- IV-E FINN compiler setup . . . 4
  - IV-E1 Docker containers . . . 4
  - IV-E2 Pypy bitstring . . . 4
  - IV-E3 Rebuilding FINN examples . . . 4
  - IV-E4 bitfiles generation from Vivado IP files . . . 4
  - IV-E5 Troubleshooting and alternative setup . . . 4

V Workflow 5

VI Results and discussion 5

VII Possible applications 6

VIII Further Implementations 6

IX Conclusions 6

References 7

List of Figures

1 FPGA acceleration: a simplified view . . . 2
2 BlueTile top and bottom view . . . 2
3 BlueTile programming board . . . 2
4 PYNQ-Z2 overview . . . 2
5 ASUS USB-BT400 Bluetooth adapter . . . 3
6 Overall flow of the project . . . 5

System on Chip Architecture PYNQ-Z2 voice commands detection Project Report

***Abstract*—The Internet of Things is driving a new technological revolution using wirelessly connected devices to sense and actuate in many use cases of our daily lives. One of the most promising applications by far is voice commands execution.**
This report presents a detailed description of the project "PYNQ-Z2 Voice Commands Detection", aiming at implementing an audio data stream between a low-power Wireless Sensor Node for data acquisition and a Central Processing Node able to analyze and classify audio samples, in order to perform voice commands detection.
With the exponential growth of connected devices, special attention should be given to the power consumed by these systems. For this reason, the transmission is performed via Bluetooth Low Energy. The goal is to see if the Bluetooth Low Energy transmission allows us to perform the desired operations without an excessive impact on the quality of results.
Audio samples are processed by a Quantized Neural Network, which is able to classify them and identify some specific commands. Results show that if the sampling pool is composed of simple words, and the audio samples are recorded in a quiet room, it is possible to reach an accuracy of around 80%, maintaining a low power consumption.

I. Introduction

Among the various applications Neural Networks (NN) have nowadays, voice recognition is one of the most promising. Voice recognition is the ability of a machine or program to receive and interpret dictation or to understand and perform spoken commands. This rising technology opens new possibilities for hands-free interaction with our digital devices.
In this report, we propose a brief explanation of the hardware we used and the setup we built to implement a voice command detection algorithm using a Quantized Neural Network (QNN), a type of network with a reduced precision of the weights, biases, and activations to consume less memory.
The goal of this project is twofold: first, implement a firmware on a small low-power Wireless Sensor Node (WSN) for data acquisition, compression, and wireless transmission using Bluetooth Low Energy (BLE). Second, implement applications running on a Central Processing Node (CPN), composed of both a Central Processing Unit (CPU) and a Field Programmable Gate Array (FPGA).
The applications will run on the CPU, and use the FPGA for data analysis speed-up. Three applications will run simultaneously on the CPN:

- An application for audio data reception using BLE and data decompression.
- An application that uses FPGA accelerated algorithms based on a QNN to detect voice commands on the received audio data.
- An audio stream that connects both applications and allows for the real-time analysis of the collected data.

The report is organized as follows: Section II discusses all the preliminary information needed to deeply understand the proposed hardware, the connection between different hardware components, and how the neural network we used works. Section III describes the hardware components adopted during this project. Section IV describes all the issues we faced in the early stages of this project, and proposes solutions to solve them. Section V presents the overall workflow for the project. Section VI shows the obtained results, and discusses their reliability. Section VII proposes some interesting use cases and scenarios. Section VIII illustrates open challenges and ideas for the next future. Section IX proposes concluding remarks.
All the related code, files and documentation can be found on the Github official repository of the project [1].

II. Background

A. Bluetooth Low Energy (BLE)

Bluetooth is a short-range wireless technology standard that is used for exchanging data between fixed and mobile devices over short distances.
Bluetooth Low Energy (BLE) is an exciting new technology that was introduced in 2010. It is attractive to consumer electronics and Internet-connected machine manufacturers because of its low cost, long battery life, and ease of deployment. As normal Bluetooth does, BLE exploits the 2.4GHz band, but it stays quiescent until an exchange of data is requested. It also uses fewer channels, a lower bit-rate and around $\frac{1}{10}$ of the maximum power with respect to traditional Bluetooth [2].
From thermometers and heart rate monitors to smart watches and proximity sensors, BLE facilitates infrequent short-range wireless data communication between devices, powered by nothing more than a dime-sized battery.
In our case, we used BLE to establish a connection between the WSN and the CPN and exchange audio data for a long period of time, using only one CR2032 battery for the whole duration of the project.

B. Machine learning & voice recognition

Machine learning algorithms build a model based on sample data, known as training data, in order to make predictions or decisions without being explicitly programmed to do so [3]. Machine learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition, agriculture, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to

perform the needed tasks.
For this project, we used the KeyWord Spotting (KWS) algorithm, trained on the Google Speech Commands v2 dataset. This method turns audio waveforms (.*wav* files) into 2D images with one channel and then classifies them to recognize the word they contain.

C. FPGA accelaration

FPGAs are able to handle compute-intensive, deeply pipelined, massively parallel operations in a much faster way with respect to a CPU. The developer can customize the FPGA architecture to his needs, using custom datapaths, custom bit-widths, and custom memory hierarchies.
The idea behind FPGA acceleration is to accurately program the FPGA so that the computation can be parallelized and executed faster (fig. 1).
In the case of algorithms allowing for parallelization and composed of many similar tasks, FPGA acceleration is able to increase the performance by 20 to 80 times [4].

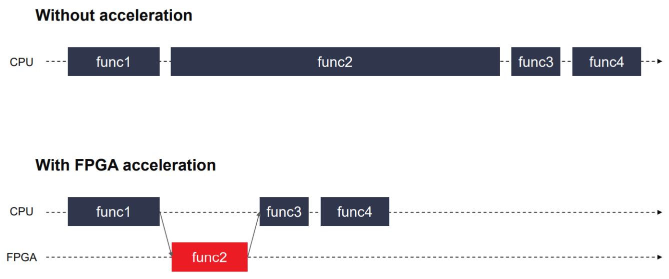

Fig. 1. FPGA acceleration: a simplified view

D. FINN framework

FINN is an experimental framework from Xilinx Research Labs to explore deep neural network inference on FPGAs, generating dataflow-style architectures customized for each network to obtain a highly-efficient implementation. The framework is fully open-source to give a higher degree of flexibility, and it is designed to target QNNs [5].
In our project, we used the framework to train and compile our QNN for the KWS algorithm.

III. Hardware components

A. Wireless Sensor Node (WSN)

The WSN used is the STMicroelectronics STEVAL-BCN002V1B [6], also known as BlueTile. This device has everything needed to fulfill our requirements:

- BlueNRG-2: BLE single-mode system-on-chip.
- MP34DT05TR-A: MEMS audio sensor omnidirectional digital microphone.
- STSW-BLUETILE-DK SDK with demo firmware including the BlueVoice library which applies ADPCM compression on a PCM audio stream and then performs half-duplex streaming of compressed data over a wireless BLE link.

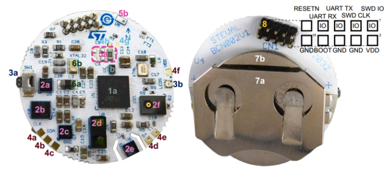

Fig. 2. BlueTile top and bottom view

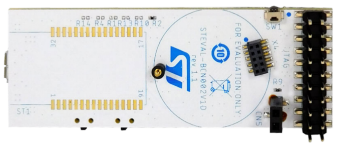

Fig. 3. BlueTile programming board

B. Central Processing Node (CPN)

The CPN used is the TUL PYNQ-Z2 developer kit [7], composed of:

- 650MHz dual-core Cortex-A9 processor
- High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
- Programmable logic equivalent to Artix-7 FPGA
- 512MB DDR3 with 16-bit bus @ 1050Mbps
- MicroSD slot (used for flashing the firmware)
- Switches, Push-buttons, and LEDs
- Line-in with a 3.5mm jack

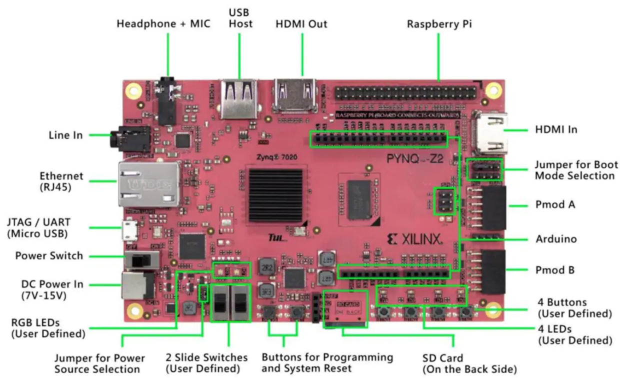

Fig. 4. PYNQ-Z2 overview

C. Bluetooth adapter

For this project, our purpose is to send audio streams from the WSN to the CPN via BLE in order to perform real-time operations while keeping a low power consumption, but the PYNQ-Z2 lacks a Bluetooth receiver. To overcome this problem, we use the ASUS USB-BT400 [8], which is one of the most advanced USB Bluetooth adapters currently on the market. Due to its high portability, ultra-small design, and low power consumption, the ASUS USB-BT400 perfectly fits our needs.

Fig. 5. ASUS USB-BT400 Bluetooth adapter

IV. Preliminary setup

A. *PYNQ environment setup*

The first thing needed to begin this project is to provide the PYNQ-Z2 with an Operating System (OS) image. At a first glance, this seemed an easy task, accomplished by simply flashing a pre-existing image into the board. The expert reader may in fact know that Xilinx provides pre-compiled OS images, and this is indeed true. However, those images are lacking the `btusb` Linux kernel module which is crucial for our application since it enables communication with Bluetooth peripherals.
Unfortunately, adding this module is harder than expected, and requires an ad-hoc build environment that we are now going to explain in detail.
To perform this task, we highly recommend using a Virtual Machine (VM), since some of the `sudo` commands used in the scripts may corrupt the host computer distribution. In the following, we will target a specific image, namely v2.6.0, since later versions do not support the FINN Framework.
So no excuses, the build begins.

1) VM and build environment setup: For this purpose, it is useful to carefully read and follow the PYNQ SD Card image Guide available in the Pynq official documentation [13]. Since targeting v2.6.0, the recommended host OS is Ubuntu 18.04 Bionic Beaver. If you are using a VM as recommended, we also suggest giving a healthy dose of virtual disk (300GB should get the work done), to avoid further problems during the compilation. In the setup of your virtualization tool give the maximum amount possible of resources to the VM, since they will boost the compilation process. Once the Ubuntu distribution is up and running, it is time to install the Xilinx Toolchain [14], whose version of interest is 2020.1.
In particular, we need:

- *Xilinx Vitis HLS Suite*: The Vitis High Level Synthesis (HLS) tool allows users to easily create complex FPGA algorithms by synthesizing a C/C++ function into RTL. It is tightly integrated with both the Vivado Design Suite for synthesis and place & route and the Vitis unified software platform for heterogeneous system designs and applications [9].
  We used the latest version, the 2022.1 [10].
- *Petalinux*: Another necessary tool, based on Yocto [11], that automates the creation of an Embedded Linux system based on the customized hardware in Xilinx FPGAs and SoCs [15].

All the required software is publicly available after signing up on the Xilinx official website [16].

2) PYNQ-Z2 image build: Once the tools are installed, it is worth mentioning that each time a new *terminal* is opened in Ubuntu to start a new build it is necessary to call the environment setup scripts that are available in each tool's installation folder. The steps needed to build the OS image are:

- Download the pre-built OS image and `rootfs` [17]. This step is useful to reduce build time, but not essential since the scripts will compile everything from scratch if these files are not provided.
- Clone the PYNQ-Z2 GitHub repository [12], open it inside a terminal and be sure to *checkout* the right version corresponding to the v2.6.0 image.
- Once this is done, a fundamental step arrives which is telling Petalinux to enable the installation of the Bluetooth kernel modules by means of a specific `.cfg` file, which is available in the GitHub repository of this project [1].
- Finally, the `make` command with the `PYNQ-Z2` option (i.e. build for the target board) can be run. The whole process takes at least 2 hours on a 4-core 8-threads laptop with 16GB of RAM available, so be warned that based on the host computer specs this time may widely fluctuate.

B. *PYNQ-Z2 image flash*

With the Linux distribution ready, it is now time to flash it into an SD card. The card should have at least 16GB of storage capacity.
For this task, we use Rufus, an open-source utility that allows the creation of a bootable USB flash drive. Once the process completes, the SD card can be inserted into the PYNQ-Z2 board and the OS can be booted.
Just a small note, be sure to place the boot jumper in the correct (SD indeed) position before powering the board up.
These are the main basic steps to flash the image, additional details are available on the official PYNQ-Z2 Setup Guide [18].

C. *PYNQ-Z2 Bluetooth setup*

The PYNQ-Z2 does not provide Bluetooth connectivity natively. However, one might use USB or Raspberry Pi interfaces to implement this feature. Our project implementation exploits the ASUS USB-BT400 USB Bluetooth adapter. To make this component correctly communicate with the `btusb` kernel module, it is sufficient to install the proprietary driver which can be easily found online [19].

D. BlueTile setup

The next step of this project requires programming the BlueTile sensor board with the firmware provided by Andrea Pignata [20] for sampling the audio signal coming from the microphone, compressing the samples in *.csv* files, and sending it by means of BLE to the PYNQ-Z2. Several steps are needed and are described in the following.

1) Build process for BlueTile: It is necessary to build the firmware to obtain an image for the BlueTile too, and for this purpose it is possible to use the ARM Keil IDE which is freely downloadable from the dedicated website [21].
Since the firmware is made of a fair number of lines of code, the free version of the Keil will ask for a license. A free activation key is provided by ST, which can be found on the *ARM Developer* website [22]. We can then proceed by opening Andrea Pignata's project within the IDE and building it. The binary compiled file will be found in the *Release* folder of the project [1].

2) Flash process for BlueTile: The compiled BlueTile firmware now only needs to be flashed into the BlueTile memory. To do so, BlueNRG Flasher Utility from ST can be downloaded from the official website [26].
The tool will simply ask to select the firmware binary file from the Keil project folder, select the target board, and click on *Flash*. The BlueTile is now ready for audio sampling and transmission.

E. FINN compiler setup

The KWS algorithm was built to run on the PYNQ-Z1 architecture; hence, it needs to be recompiled and adapted to the PYNQ-Z2 through the help of the FINN compiler [27]. To perform such an operation, a set of preliminary steps are required.

1) Docker containers: FINN runs inside a Docker container, and it comes with a script (named *run-docker.sh*) to build and launch a dedicated one; however, we must configure docker to run without root since the Docker daemon binds to a Unix socket which is normally owned by root. The configuration can be easily done by creating a Unix group called `docker` and adding users to it [23].
It is mandatory to set two environmental variables inside the script before running it [24]:

- `FINN_XILINX_PATH`: pointing to the Xilinx tools installation on the host;
- `FINN_XILINX_VERSION`: specifying the Xilinx tools version to be used (e.g. 2022.1).

2) Pypy bitstring: A further requirement for a correct configuration of the compiler is the installation of *bitstring*, a pure Python module that handles the creation, manipulation and analysis of binary data [25].

3) Rebuilding FINN examples: The already existent KWS algorithm must be adapted to run on the PYNQ-Z2 board, and the related bitfiles and Vivado IP files need to be regenerated; to do so we first clone FINN at the right commit, checking that the requirements are met; further, the above mentioned environmental variables should be properly set to execute the *run-docker.sh* script [27].
Each FINN-example comes with a *build.py* script, describing the configuration adopted to compile the addressed example, which also requires the target pre-trained MLP ONNX models and pre-processed validation data. Such files are downloadable through a simple script, available on the official GitHub repository of the finn-examples [28].
We wrote a bash script to automate the generation of the IP files and bitfiles: the script specifies the absolute path of the KWS examples on the local machine and launches the *run-docker.sh* script, providing the path as an argument.
At the end of the build, all the output files are available in a dedicated directory; in order to target the PYNQ-Z2 board we are not interested in the bitfile itself but in the IP files, which must be integrated into a Vivado IP design using an Advanced eXtensible Interface (AXI) stream.

4) bitfiles generation from Vivado IP files: Vivado HLS tool offers also the possibility of generating bitfiles starting from scratch, or creating custom PYNQ overlays, by using IP designs for a wide range of applications [29].
To begin integrating an IP file inside the PYNQ-Z2 a block design needs to be created inside a new Vivado project; the IP design must be visible to the tool, so the repository containing the files should be added to the project settings.
Proceeding with the flow, we add the required blocks:

- The Zynq Processing system;
- Our HLS IP;
- The Processor System Reset block;
- AXI streams and interfaces to glue the other blocks together.

After the creation of the block design we must validate it; if the validation completes without errors the next step consists in creating the HLS wrapper of the block design so that Vivado can generate the bitstream.
A useful feature offered by Vivado is the possibility of generating a *.tcl* script which can be executed to replicate the block design from scratch.
Once the bitfiles and the *.tcl* script have been generated, give them the same name and copy them into the target directory of the PYNQ-Z2 board.

5) Troubleshooting and alternative setup: Several issues were experienced during the implementation of the project, which led to the development of an alternative, working solution.
A first attempt has been done with the provided files, without any modification regarding the board configuration. The HLS IP file has been used as input for a new block design in Vivado, and the generation of the bitstream file was completed without major problems; however, some problems arose when running the KWS example. In particular, the issues were related to the overlay, the union of the bitstream, and the project block diagram tcl file. These errors were probably due to the bitfile requiring a module which has not been inserted during the development of the block design.
Another bitfile generation has been performed, this time changing the content of the *build.py* script related to the KWS algorithm; in particular, we changed the platform name to ”Pynq-Z2” in the configuration section. The resulting bitfile was then loaded in the *bitfile* directory on the Pynq-Z2 but the

same exception occurred when trying to run the example.
As the last experiment, we decided to take the bitfiles related to the KWS network built for the PYNQ-Z1 board and load them into the PYNQ-Z2, trying to execute the example contained in the jupyter_notebooks folder. By providing both the *.bit* and the *.hwh* file into the board we succeeded in running the example.
To achieve this result the Google Speech Commands dataset must be downloaded since it constitutes the foundation of the neural network and has also been used for its training.
This dataset is contained in NumPy files (*.npy*), which is a standard binary file format for persisting a single arbitrary NumPy array on a disk. The above mentioned files are available in *.zip* format on the GitHub repository of the FINN-examples, inside the "data" folder [30].
The dataset files must be moved into the "data" directory, which has been created inside our PYNQ-Z2 board under a specific path[1].
The KWS algorithm is not available for the PYNQ-Z2 board, so the bitfile and the Hardware handoff (*.hwh*) file must be downloaded and placed inside the *bitfile* directory[2]; to achieve a successful configuration the two files must have the same name.
As the last step, the example has been launched by connecting to jupyter notebooks via Ethernet link, navigating to the *finn_examples* folder and then to the *4_keyword_spotting.ipynb* file.

V. Workflow

Initially, one may want to verify the correct operation of the BlueTile Bluetooth connection through the mobile app for Android devices *ST BLE Sensor demo* [31] by STMicroelectronics: the app provides an easy-to-use GUI for checking the behavior of all the sensors of the BlueTile, including the digital microphone.
The first step is to establish the Bluetooth connection between the BlueTile and the PYNQ-Z2. With the newly built Pynq image, the `btusb` kernel module and the Bluetooth driver installed, this step is very straightforward. It is in fact sufficient to launch the `multiple_client_bluepy.py` Python script on the PYNQ-Z2 to establish the connection and to start capturing the audio sample, which is directly saved on the board. The original BlueTile Python client has been properly modified by Andrea Pignata [20] in order to avoid timestamp processing and increase the audio quality, and is available on the GitHub repository of the project [1] as well. On top of that, we had to write our own *time_ns()* function because it is not natively supported by Python3.6 but only available in newer versions (i.e. since Python 3.7).
With this done, we can proceed with the recording of our samples in *.csv* format. This is a text file that has a specific format allowing data to be saved in a table-structured format. However, the *.csv* extension is not supported by the QNN, so we used a Python script to convert it into a more common *.wav* audio file, namely `convert_upsampling.py` available on GitHub [1]. During the conversion, up-sampling from 8kHz to 16 kHz is also performed to artificially increase the audio quality. The up-sampling is obtained by taking two original samples values and inserting their average in between them. Although not necessary, it has proven a pretty effective method to improve the network hit rate. This is due to the fact that the maximum sampling rate of the BlueTile is 8 kHz which is half of the rate of the samples that have been used to train the neural network, 16 kHz indeed. Particular care must be taken during this procedure: audio signals must be of *mono* type, otherwise they will not be recognized by the neural network.
As a final step, the audio file and the dataset are given in input to the QNN, which performs the classification. It is needed to specify the directory containing the samples inside the *Jupyter notebooks example* ([34], section "Loading and pre-processing the audio file"). Based on the ratio between the correctly classified samples and the overall number of samples, we can obtain the accuracy of the system.

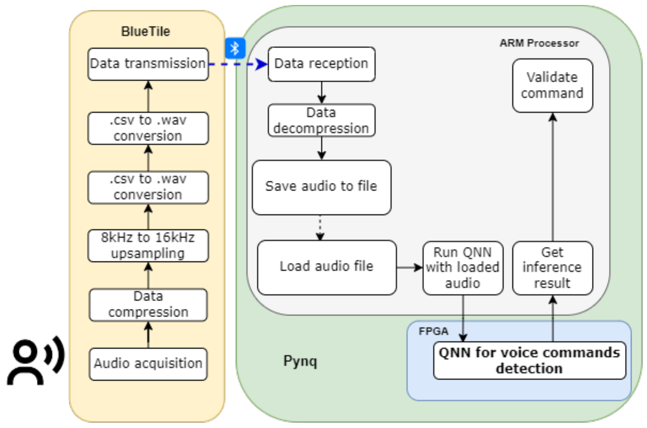

Fig. 6. Overall flow of the project

VI. Results and Discussion

To test the commands recognition performed by the QNN, we tried all the possible commands that the model is able to classify: "down", "go", "left", "no", "off", "on", "right", "stop", "up" and "yes".
In the first round of testing, the obtained accuracy was fairly low settling on a 30% to 50%. What we noticed from these attempts is the problem derived from using the microphone at a distance higher than one meter from the board; if this was the case, audio samples turned out to be distorted. Indeed, we had to make our test very close to the PYNQ board to achieve acceptable audio quality.
In addition, the commands were not always recognized on the first try, but sometimes they seemed hard to be identified, maybe because of the different pronunciations and cadences used in order to deeply test the model.

[1]the absolute path is: /usr/local/lib/python3.6/dist-packages/finn_examples/data/
[2]The absolute path is: /usr/local/lib/python3.6/dist-packages/finn_examples/bitfiles/Pynq-Z2

However, we observed that by placing close to each other the BlueTile and the PYNQ-Z2 board while recording, together with applying an upsampling process to the *.csv* audio files, our records were much more clear and easy to be classified by the network, pushing the obtained accuracy up to a solid 80%.

VII. Possible applications

Despite the limited set of voice commands that the network model is able to recognize, many possible implementations for it could be found. The biggest advantages of our proposed solution are the low power consumption and the embedded deployment, which make it reusable in many different fields. The QNN is in fact adaptable to systems that do not provide a high amount of memory, while the BLE is especially suited for all those systems that cannot be recharged continuously but need to operate for long periods.

A possible application could be in the automotive field as an in-car speech recognition system. Indeed, numerous types of research [32] have proven as the global automotive voice recognition system market is growing in response to the increasing popularity of AI technologies.

Drivers can use voice assistants built into their cars for common actions like initiating a phone call, selecting radio stations, setting up directions, or playing music while also allowing them to keep their eyes on the road and hands on the wheels.

Many more applications could be further investigated like home assistance or healthcare.

VIII. Further Implementations

In this section, we list all the possible improvements we considered while working on this project but not implemented due to a lack of time, knowledge, or resources:

- The adopted NN is able to identify a finite set of commands, thus limiting the degree of freedom. Besides this, the Google speech commands dataset is intended as a teaching example [33]; this opens the possibility of using other commercial, more complex datasets or building customized datasets. FINN compiler also offers the possibility of training a NN from scratch, making possible the recognition of very specific commands or sentences. In this last case, the compiler works together with *Brevitas* [35], a training library created by Xilinx and based on the *Torch* library.
- Regarding the KWS example, the possibility of extending the maximum duration of an audio track (currently limited to 1 second) could be a further enhancement, because it could open to training and recognition of longer, more complex words.
- Another interesting addition would be to make the PYNQ-Z2 board react to the recognized commands in order to perform some actions in response (e.g. turning a LED ON/OFF).
- To go on, also the hardware equipment could be investigated for strengthening purposes: by using a more powerful microphone and/or Bluetooth receiver the user could produce better samples or he could record them at a greater distance from the receiver, limiting possible environmental interferences.
- Finally, the environment could be further improved by enabling Real-Time operations. For this purpose, we imagine the board passive for most of the time, but able to react as soon as a voice command is sent, without the need of performing additional manual commands. To reach this goal we assessed the possibility of recording longer samples and then fragmenting them into multiple frames. Unfortunately, this solution turned out to be unfeasible due to the long elaboration time required by the conversion and upsampling processes.

IX. Conclusions

Thanks to this project, we explored many topics that are on the crest of a wave nowadays.

We learned how to connect different devices through a Bluetooth connection, how to build an OS image from scratch, how NNs work, and how to use them. We believe that the results obtained are satisfying and that the huge project setup enabled us to replicate the workflow in a very smooth and fast way.

We are sure that the experience gained through this project will be very useful in the later stages of our working career.

In conclusion, we cannot deny that this project was hard and made us feel in the middle of nowhere many times, however, after several months and several tries, we believe that we succeeded in our goals and all the failures let us grow as engineers both in terms of hard skills and soft skills.

References

[1] Official Github repository for the ”PYNQ-Z2” project [Online]. Available: https://github.com/SoC-Arch-polito/pynq22

[2] Wikipedia page on Bluetooth [Online]. Available: https://en.wikipedia.org/wiki/Bluetooth[Accessed:Apr. 21,2023].

[3] Wikipedia page on Machine Learning [Online]. Available: https://en.wikipedia.org/wiki/Machine_learning[Accessed:Apr. 21,2023].

[4] How FPGA acceleration works [Online]. Available: https://www.xilinx.com/publications/events/developer-forum/2018-frankfurt/fundamentals-of-fpga-based-acceleration.pdf[Accessed:Apr. 21,2023].

[5] “FINN - Fast, Scalable Quantized Neural Network Inference on FPGAs”[Online]. Available: https://github.com/Xilinx/finn

[6] STEVAL-BCN002V1B BlueTile - Bluetooth LE enabled sensor node development kit [Online]. Available: https://www.st.com/en/evaluation-tools/steval-bcn002v1b.html

[7] “PYNQ- Python Productivity for Zynq”[Online]. Available: https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html[Accessed: Apr. 21,2023].

[8] USB 4.0 Bluetooth adapter [Online]. Available: https://www.asus.com/it/networking-iot-servers/adapters/all-series/usbbt400/[Accessed:Apr. 21,2023].

[9] Vitis HLS software platoform - Overview [Online]. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html[Accessed:Apr. 21,2023].

[10] Installing the Vitis Software Platform [Online]. https://docs.xilinx.com/r/2022.1-English/ug1400-vitis-embedded/Installing-the-Vitis-Software-Platform[Accessed:Apr. 21,2023].

[11] YOCTO project [Online]. Available: https://www.yoctoproject.org[Accessed:Apr. 21,2023].

[12] PYNQ-Z2 Github repository [Online]. Available: https://github.com/Xilinx/PYNQ[Accessed:Apr. 21,2023].

[13] Official PYNQ SD Card image documentation - ReadTheDocs [Online]. Available: https://pynq.readthedocs.io/en/latest/pynq_sd_card.html [Accessed: May 03,2023]

[14] Vitis Core Development Kit v2020.1 Download - Xilinx [Online]. Available: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/archive-vitis.html [Accessed: May 03,2023]

[15] PetaLinux Tools Documentation - Xilinx [Online]. Available: https://docs.xilinx.com/v/u/2020.1-English/ug1144-petalinux-tools-reference-guide [Accessed: May 03,2023]

[16] Xilinx [Online]. Available: https://www.xilinx.com/ [Accessed: May 03,2023]

[17] PYNQ Prebuilt Image v2.6.0 - PYNQ Forum [Online]. Available: https://discuss.pynq.io/t/prebuilt-board-agnostic-root-filesystem-and-prebuilt-source\protect \discretionary {\char \hyphenchar \font }{}{}distribution-binaries-for-version-2-6-1/5127 [Accessed: May 03,2023]

[18] Official PYNQ-Z2 Setup Guide - ReadTheDocs [Online]. Available: https://pynq.readthedocs.io/en/latest/getting_started/pynq_z2_setup.html [Accessed: May 03,2023]

[19] Linux Installation guide for Asus USB-BT400 - GitHub [Online]. Available: https://gist.github.com/ssledz/69b7f7b0438e653c08c155e244fdf7d8 [Accessed: May 03,2023]

[20] Andrea Pignata - GitHub [Online]. Available: https://github.com/andreapignaz [Accessed: May 03,2023]

[21] ARM KEIL - Download products [Online]. Available: https://www.keil.com/download/ [Accessed: May 03,2023].

[22] ARM Developer - ST License [Online]. Available: https://developer.arm.com/documentation/kan344/1-1/License [Accessed: May 03,2023].

[23] Manage Docker as a non-root user [Online]. Available: https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user[Accessed:Apr. 21,2023].

[24] FINN examples - Getting started [Online]. Available: https://finn.readthedocs.io/en/latest/getting_started.html#quickstart [Accessed: May 03,2023].

[25] bitstring 4.0 documentation [Online]. Available: https://bitstring.readthedocs.io/en/stable/[Accessed:Apr. 21,2023].

[26] [Online]. Available: https://www.st.com/en/embedded-software/stsw-bnrgflasher.html [Accessed: May 03,2023].

[27] Rebuilding finn-examples [Online]. Available: https://github.com/Xilinx/finn-examples/tree/main/build[Accessed:Apr. 21,2023].

[28] KeyWord Spotting example - Pretrained model and data [Online]. Available: https://github.com/Xilinx/finn-examples/tree/main/build/kws/models[Accessed:Apr. 21,2023].

[29] PYNQ - Overlay tutorial [Online]. Availble: https://pynq.readthedocs.io/en/v3.0.0/overlay_design_methodology/overlay_tutorial.html [Accessed: May 03,2023].

[30] Google speech dataset for KeyWord Spotting example [Online]. Available: https://github.com/Xilinx/finn-examples/blob/main/finn_examples/data/all_validation_kws_data_preprocessed_py_speech.zip.link[Accessed:Apr. 21,2023].

[31] ST BLE Sensor App - Google Play [Online]. Available: https://play.google.com/store/apps/details?id=com.st.bluems&hl=en_US&pli=1[Accessed:Apr. 21,2023].

[32] ”Automotive Voice Recognition System Market (By Technology: Embedded, Cloud-Based, and Hybrid; Application: AI and Non-AI; By Vehicle Type: ICE and BEV) - Global Industry Analysis, Size, Share, Growth, Trends, Regional Outlook, and Forecast 2022 – 2030”, Dec. 2021. [Online]. Available: https://www.precedenceresearch.com/automotive-voice-recognition-system-market.

[33] Launching the Speech Commands Dataset [Online]. Available: https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html [Accessed:Apr. 24,2023].

[34] KWS example - source code [Online]. Available: https://github.com/Xilinx/finn-examples/blob/main/finn_examples/notebooks/4_keyword_spotting.ipynb [Accessed: May 03,2023].

[35] Brevitas - PyTorch library for neural network quantization [Online]. Available: https://github.com/Xilinx/brevitas [Accessed:Apr. 24,2023].