SUP'COM

المدرسة العليا للمواصلات بتونس

ÉCOLE SUPÉRIEURE DES COMMUNICATIONS DE TUNIS

Telecommunications Engineering Curriculum

Graduation Internship Report

Combining FPGA and Edge-to-Cloud solutions to build AI-based self-adaptive applications

Realized By:
Abir Chebbi

Academic Supervisor :
Prof. Rached Hamza

Professional Supervisor :
Prof. Nabil Abdennadher
Francisco Mendonça

Work proposed and fulfilled in collaboration with:

h e p i a

Haute école du paysage, d'ingénierie
et d'architecture de Genève

Academic year : 2021 - 2022

Higher School of Communication of Tunis
SUP'COM
2083 Communications Technology City - Elghazela - Ariana - Tunisia
Tel. +216 70 240 900 - website: www.supcom.mincom.tn

Signatures

Mr. Nabil Abdennadher

Mr. Rached Hamza

DEDICATION

This work is dedicated to my beloved parents, Ali & Souad, for all the support and encouregment they gave me.

To all my family, I dedicate this work as a symbol of my love and eternal thanks.

ACKNOWLEDGMENTS

In term of this project, I would like to express my heartfelt gratitude to anyone who contributed to the development of this work.

I would especially like to thank my supervisor, Mr. **Nabil Abdennadher**, for his guidance and constant supervision as well as for entrusting me with missions that challenged my knowledge.

I would also like to thank **Francisco Mendonça**, **Raoul Dupuis** and **Poleggi Marco Emilio** for all the explanations and help throughout this project .

My sincere thanks go to **Quentin Berthet** and **Gabriel Da Silva Marques** for their kind cooperation and their support.

Finally, I would like to thank my supervisor Mr.**Rached Hamza** for his continuous support and his remarks.

ABSTRACT

Noise pollution, like any other pollution, is a nuisance to society. It can be found anywhere but especially in cities. Studies show that anything at or above 55 decibels can trigger an increase in blood pressure, heart rate, and stress hormones in the blood. Road traffic remains the main source, where it represents 80% of this noise pollution followed by railways, airports, and industry.

In this context, the “Noise Radar” project comes to offer a generic solution for road traffic monitoring by developing intelligent acoustic sensors able to recognize noise activities for various vehicle classes and crack down on the vehicles making too much noise.

Therefore, we intend, in this project, to implement a self-adaptive Edge-to-Cloud platform for managing a network of Noise Radar sensors at a city scale. The Edge-to-Cloud architecture provides a coordination mechanism that allows us to monitor and update Noise Radar sensors with minimal external intervention. Through this network, we aim to deploy inference modules to be executed on the edge, that were trained on the cloud. As a result, Field Programmable Gate Array (FPGA) can be utilised for this, since they can execute instructions in parallel and for their security level also.

As part of our contribution to this project, we conducted research to compare well-known edge-to-cloud technologies and choose to test two of them in this project. The workflow for developing an FPGA-accelerated application was then integrated. In a subsequent step, we containerized this application and tested its deployment using edge-to-cloud technologies.

As a result, we discovered the ability to update the FPGA via the cloud, which reinforces the concept of self-adapting edge devices. As a result, better models will be generated, and we will be able to update the AI models running on the edge.

Keywords: Edge, IoT, Cloud, Noise pollution, Field Programmable Gate Array

LIST OF ABBREVIATIONS

FPGA : Field Programmbale Gate Array

NoRa : Noise Radar

NN : Neural Network

AI : Artificial Intelligence

SONAL : SouNds AnaLytics

MQTT : Message Queuing Telemetry Transport

OS : Operating System

MPSOC : Multiprocessor On Chip

TPU : Tensor Processing Unit

DPU : Deep Processing Unit

CNN : Convolutional Neural Network

HTTP : Hypertext Transfer Protocol

CPU : Central Processing Unit

PAC : Platform Assets Container

XRT : Xilinx RunTime

CONTENTS

General Introduction **1**

1 Presentation of Securaxis' software **5**
1.1 Introduction 5
1.2 SONAL software 5
1.3 SONAL software accelerated on FPGA 8
1.4 Improve the performance of SONAL application 9
1.5 Conclusion 10

2 Comparative study between Edge-to-Cloud solutions **11**
2.1 Introduction 11
2.2 Balena 14
2.3 Google Cloud Platform 16
2.4 Azure IoT Edge 17
2.5 NuvlaEdge 18
2.6 AWS GreenGrass 19
2.7 Conclusion 20

3 FPGA accelerated application **21**
3.1 Introduction 21
3.2 Field Programmable Gate Arrays 22
3.3 Zynq MPSoC hardware overview 23
3.4 Vitis 25
3.5 Accelerated application structure 26
3.6 Sample FPGA application 26
3.6.1 Prepare the kernel code and host code for Vector add 26
3.6.2 Prepare the Platform Assets Container 28
3.6.3 Activate the PAC on the device 28
3.7 Accelerate convolutional neural network 31
3.8 Conclusion 32

4 Deployment of FPGA-accelerated application **33**
4.1 Introduction 33
4.2 Containerize FPGA application 33
4.3 Deployment through Azure IoT Edge 36
4.4 Deployment through Nuvla/NuvlaBox 38
4.5 Deploy FFT accelerated application 42
4.6 Conclusion 43

General Conclusion 44

LIST OF FIGURES

1 Number of people in the EEA-33 member countries exposed to noise levels above 55 dB Lden, 2012, source([21]) 2

1.1 Inference model of SONAL software 6
1.2 Training model of SONAL software 7
1.3 SONAL components computed on FPGA device 9
1.4 Update SONAL application through Cloud 10

2.1 A self-adaptive SONAL application scenario 12
2.2 Balena architecture 15

3.1 The basic structure of the FPGA 22
3.2 Zynq UltraScale + MPSoC ZCU104 evaluation board 23
3.3 The base components of XCZU7EV MPSoC 24
3.4 Vitis Unified Software Platform 25
3.5 Structure of an application interacting with the FPGA device 26
3.6 Compile the Vector add on Vitis 28
3.7 Checking the current available HW configurations 29
3.8 Activate the configuration for the Vadd accelerator 30
3.9 Checking the activated HW configurations 30
3.10 Executing Vector add application on the ZCU104 device 30
3.11 Loading kernel code to PL part 31

4.1 The environment needed for the container in order to communicate with the FPGA device 34
4.2 Pass the host code and the kernel code to the container 35
4.3 Run the Vector add container 35
4.4 The Edge device based FPGA is connected to Azure IoT edge 36
4.5 Two docker containers running on the Edge device based FPGA and representing the IoT Edge Runtime 36
4.6 Vector add image stored in Azure Container Registry 37
4.7 Vector add module running on the Edge-based FPGA device 37
4.8 The workflow to deploy an application through Azure IoT Edge to the Edge-based FPGA device 38

4.9 Vector add through Azure interface 38
4.10 The Edge based FPGA device connected to Nuvla.io 39
4.11 Deploy Vector add application on NuvlaBox 41
4.12 Vadd_fpga application running in NuvlaBox 41
4.13 FFT container runing on NuvlaBox 43
4.14 FFT container running on Azure IoT Edge 43

LIST OF TABLES

3.1 The required files in PAC 29

GENERAL INTRODUCTION

Now more than ever, it has become crucial to adopt a multifaceted approach when managing the cities and smartening them. For instance, urban mobility affects the environment (e.g. air pollution, noise, CO2 emissions, biodiversity losses), the citizens (e.g. traffic chaos, security, decreased quality of life), and infrastructure investors (e.g. overloaded road infrastructures with increased maintenance and lower lifecycle).

Noise pollution is still a major public health issue. As seen in Figure 1, published by the European Environment Agency, road traffic is the most common source of environmental noise. A wide range of measures is included in the noise management action plans established by several states. The most typically stated measures in cities are those aimed at traffic control. Replacement of road surfaces, improved traffic flow, and lower speed restrictions are examples of such interventions.

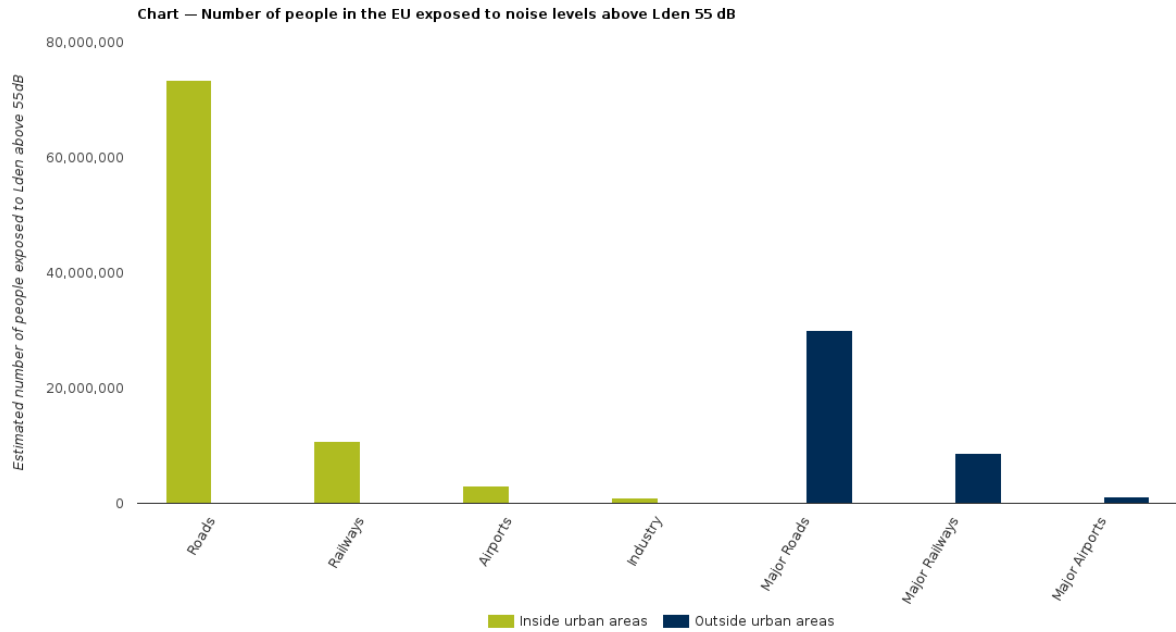

Figure 1: Number of people in the EEA-33 member countries exposed to noise levels above 55 dB Lden, 2012, source([21])

The process of assessing the level of noise in industrial and residential areas is referred to as noise monitoring. The Global Noise Monitoring Market helps to better understand patterns that may act to reduce noise pollution. It was valued at USD 621.03 million in 2016 and is expected to increase to USD 900.11 million by 2025 [11].

In response to this market trend, the Geneva based SecurAxis[25] start-up proposes a camera-less noise sensor that "catalogues" vehicles according to "classes" previously defined by the end user (noisy/non noisy vehicles, cars, trucks, buses, motorcycles, e-cars, etc.). This noise sensor supports a Deep Learning based software named SONAL and will be the backbone on which noise management digital platforms can rely to detect and identify noise vehicles. SecurAxis' solution allows 3D acoustic detection for a better understanding of the mobility patterns of cities and authorities independently of the services currently provided by GAFAM (Google, Apple, Facebook, Amazon, MicroSoft).

SecurAxis' ambition is to address the city needs in the field of sound measurements on moving vehicles. In Europe, this solution is currently being tested in Paris, London, Porto, Maastricht, Stuttgart and Oulu. In Switzerland, SecurAxis sensors are already deployed in Geneva, Fribourg and Zurich. To improve its product, Securaxis is participating in The Noise Radar (NORA) Innosuisse project with HES-SO (HEPIA) and EPFL [1]. The objectives of NORA are as follows:

1. Address the urban traffic noise pollution issue by devising acoustic sensors able to measure in real time the psychoacoustic impact of it.

2. Develop and set up a distributed edge-cloud based platform that will allow a real-time and self-adaptive detection, counting, monitoring and classification of vehicles via their emitted noise under several constraints.

3. Answer public health noise and air quality related issues by being able to detect noisy vehicles

The work presented in this document deals with the second objective. Concretely speaking, this research aims to develop a "self-adaptive" solution able to classify vehicles according to the noise they generate. The solution will be deployed on an edge-to-cloud platform. The edge device is camera-less and relies on the FPGA technology to implement the vehicle classification.

Self-adaptability relies on a coordination model, provided by the edge-to-cloud infrastructure, which allows Securaxis sensors to communicate with the cloud in order to ensure provisioning and continuous delivery to the edge. The objective of self-adaptability is that the intelligence within the edge can adapt to changing operating conditions. This could be done by improving inference modules in case of poor performance.

Edge computing is a promising approach for dealing with the issues such as delay in response, high bandwidth consumption, and reliability issues. The main idea of edge computing is to process the data at edge nodes without the need to transfer gigabytes of data to the cloud. For privacy concerns, edge computing analyses sensitive data within a private network, thereby protecting them from any malicious use. Thus, local data processing is the perfect opportunity to anonymize data or remove together any privacy intrusion aspect.

Moreover, it makes more sense to process the data close to its source, so as to transform them into valuable information. In other words, it's useless to stream all the raw data to the cloud, we only need to be notified when such "abnormal" events occur, for example, low accuracy of detection.

As specified earlier, SONAL is based on Deep Learning technology and Neural Network architecture. Using Neural Networks (NN) requires a lot of power and is computationally expensive. Moreover, having the program on the edge must be safeguarded to protect the intellectual property of a company. To overcome this problem, SONAL can be implemented on a Field Programmable Gate Array (FPGA) circuit. An FPGA-based implementation can ensure security, privacy, and an optimal balance between power consumption and performance.

This report is organised as follows: The first chapter presents the overall architecture of the solution to implement. It particularly describes the software deployed on the Securaxis noise sensor. The second chapter provides a comparative study of five candidate edge-to-cloud solutions that could be used as a backbone to implement the targeted solution. The third chapter deals with the implementation of SONAL on an FPGA infrastructure. The fourth and final chapter walks us through

the deployment of the integrated solution on two edge-to-cloud solutions: Azure and Nuvla/NuvlaEdge.

CHAPTER 1

PRESENTATION OF SECURAXIS' SOFTWARE

1.1 Introduction

Vehicles produce heat, sounds, and a magnetic field. Many approaches to vehicle detection have been investigated using various types of signals. The approach based on acoustic signals appears to be the most promising. Moving vehicles make distinct sounds. These sounds are produced by moving parts, friction, wind displacement, emissions, etc.

In this context, in this chapter we introduce the SONAL application, SONAL is an intelligent software developed for Securaxis sensors, that allows the detection and classification of vehicles based on acoustic signals.

The chapter is composed of three sections: The first section defines in general SONAL application. The second section represents the implementation of SONAL on the FPGA. The third and final section deals with a few improvements carried out to SONAL to have better accuracy.

1.2 SONAL software

SONAL is an intelligent software developed by Securaxis, which is a company that aims to develop, install and operate safety and security solutions for smart cities. This software is able to extract vehicle characteristics based on the noise coming from the traffic. These characteristics are basically the number of vehicles, type, direction, and even an estimation of the speed of the traffic. After detecting vehicles, the sound emitted is analysed and the vehicle is categorised:

1. Car
2. Motorcycle
3. bus/truck

4. bicycle
5. unknown (the vehicle is not identified)

The classification is based on a Neural Network (NNc) that recognises the vehicle according to the sound it generates (Figure 1.1). After receiving the signal from the microphone, signal processing must be performed. The data is then sent to an intelligent model (detection Neural Network), which determines whether or not the sound is generated by a vehicle. If this is the case, the output will be fed into the classification model to classify the vehicle.

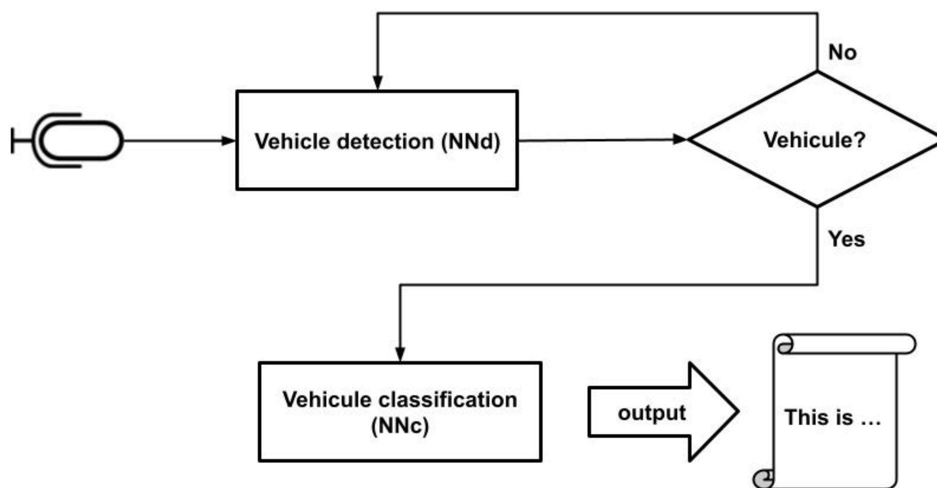

Figure 1.1: Inference model of SONAL software

During a learning step (Figure 1.2), a camera that films the vehicles allows the noises made by the vehicles to be labeled. The labeled data is then used to launch a learning algorithm. In fact, using a camera will be beneficial because it will record the passing vehicles. At the same time, it will retrieve the image-related record from the microphone to have at the end an audio/frame sampler.

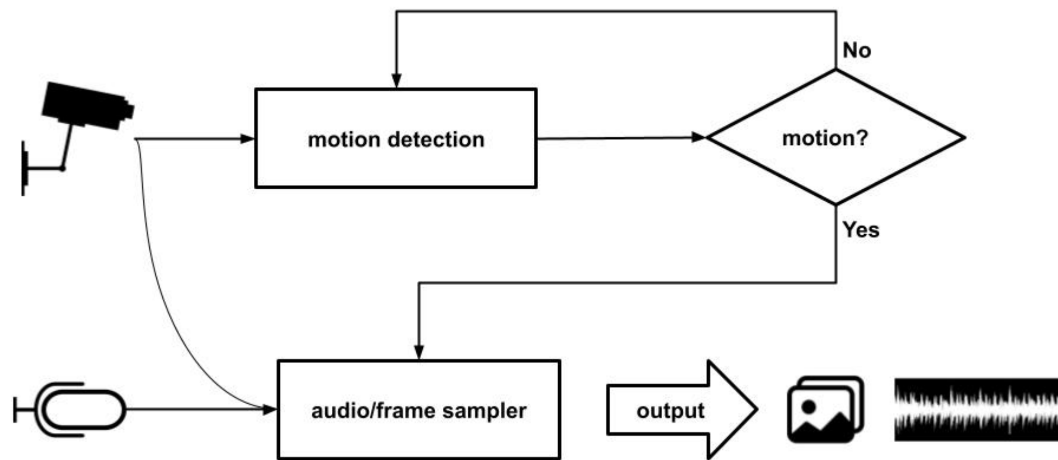

Figure 1.2: Training model of SONAL software

To summarise, SONAL is first trained to recognise vehicles using a training database of thousands of sound samples that have previously been labeled using a camera. Once trained, the software is deployed in production on acoustic sensors placed on the roadside and distributed on a city scale.

1.3 SONAL software accelerated on FPGA

As stated in the introduction. One of the goals of the NORA project is to benefit from FPGA capabilities. We intend to use FPGA accelerators to implement parts of the SONAL application that need to be accelerated.

In order to realise the advantages of an FPGA accelerated application, having a look at which parts in the code take more time to be executed and implementing these parts in custom hardware results in an ideal balance between performance and power. Therefore, a profiling process of SONAL software has to be done.

The two intelligent components NNd and NNc (Figure 1.3) in the SONAL application are essentially heavy computations on the Edge. Using a CPU for this kind of computation takes a lot of time. In addition to the NN algorithm, there are other modules of SONAL that must be accelerated such as Fast Fourier Transform (FFT), which is one of the essential building elements of Digital Signal Processing (DSP) and Signal Analysis and it represents an important part of the SONAL application. FFT helps to process signals in the frequency domain. Standard processors might be too slow for this kind of computation. Matrix multiplication has been considered also a heavy computation.

Figure 1.3 depicts the main components that must be accelerated on the FPGA: This diagram has been heavily abstracted in order to facilitate understanding of FPGA-accelerated applications. FFT and matrix multiplication are two specific functions that can be accelerated using hardware accelerators. However, there is a general accelerator on the FPGA called Deep-learning Processor Unit (DPU) that is more generic to Deep Learning algorithms for the NN components. The third chapter will provide further details related to this part.

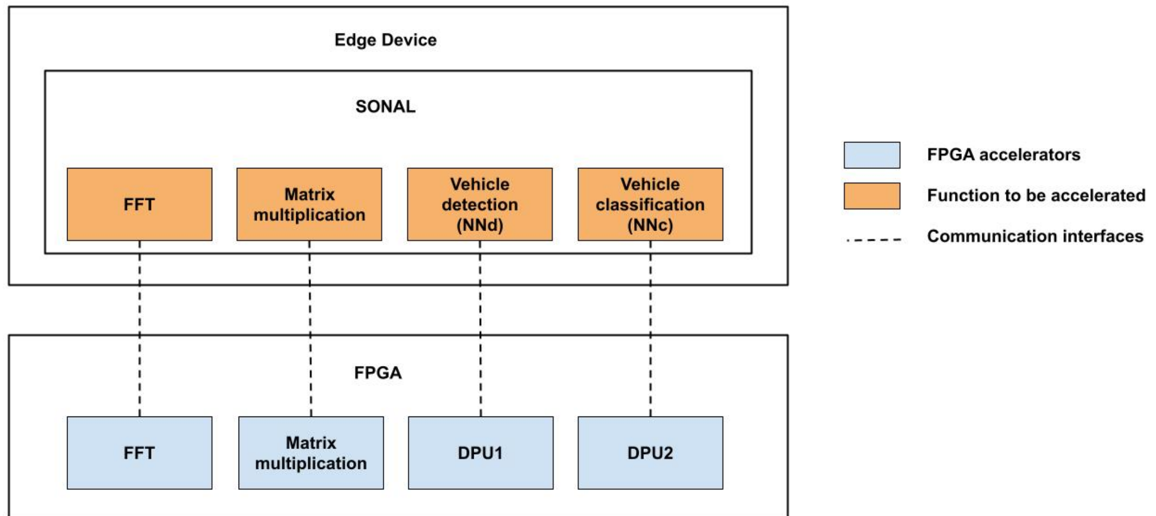

Figure 1.3: SONAL components computed on FPGA device

1.4 Improve the performance of SONAL application

As we all know, model performance is a measure of how well an AI model performs. However, it is not solely about the model's accuracy. Model performance is an evaluation of the model's ability to perform a task accurately. We are not just talking about training data, but also about the runtime data when the model is deployed. As a result, it is necessary to evaluate model performance in real-time to correct our model.

In our case, the edge's classification performance depends on the "context" where the device is deployed, so we are sure that the model will not have the same performance. This depends on the road types, street configuration, etc. Therefore, we must consider that after deployment, each sensor will have its own identity and environment.

Furthermore, the model may perform poorly in a bad environment. For example, detecting a vehicle in adverse weather conditions, particularly on a rainy day, is not the same as it is on a normal day. In this case, the detection/classification accuracy might drastically decrease.

As a solution, an adaptive Edge-Cloud platform is required. In other words, we need to ensure continuous delivery to edge devices. In fact, in case of poor performance due to isolated malfunction or contextual shift, the edge devices' inference modules are automatically improved. Only relevant notifications are sent to the cloud in this situation, ensuring the continuous transmission of edge devices' intel-

ligence.

Figure 1.4 details the scenario of self-adaptive application, on the edge part data that will be processed locally (signal processing coming through microphones and model inference) by SONAL software. On the cloud part, the AI model are built and trained, a human supervised labelling service is also needed to prepare the training data. Therefore, the traffic between the edge and the cloud is limited to an information exchange such as error notifications, a new improved version of the AI models deployed on the edge, detection results, etc.

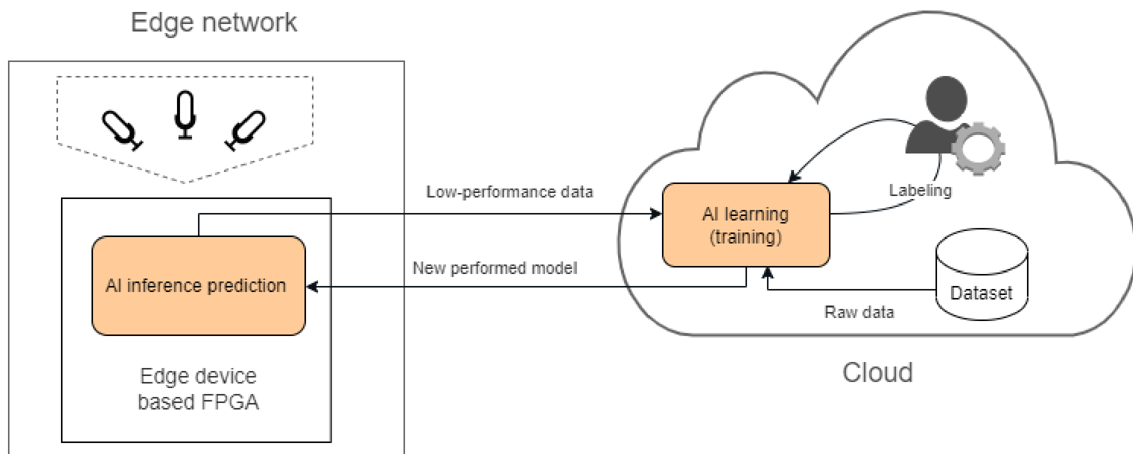

Figure 1.4: Update SONAL application through Cloud

1.5 Conclusion

This chapter was more general in nature, as we defined the software that would run on the edge sensors. We also discussed two areas where this software could be improved, and this serves as an introduction to the next chapter: The first aspect was accelerating SONAL on the FPGA, with more technical details which will be provided in the third chapter. We also present the self-adaptive aspect, which will be addressed in the second and fourth chapters.

CHAPTER 2

COMPARATIVE STUDY BETWEEN EDGE-TO-CLOUD SOLUTIONS

2.1 Introduction

Edge computing computes data close to nodes rather than sending it all to the cloud to be processed. The user migrates from the cloud to the edge for several reasons: faster processing, lower latency, data security, uninterrupted connectivity, lower traffic, lower connectivity costs, and less maintenance.
This approach does not represent a replacement for the cloud; rather, it represents an extension of the cloud. Several edge-to-cloud services stand out in this context, with the goal of reinforcing the concept of edge computing. These technologies aim to monitor the edge devices and ensure deployment of complex event processing, machine learning, etc.

As discussed in section three of the first chapter, we want to self-adapt the SONAL application and monitor our FPGA-based device via the cloud. In the long run, we want to ensure that our AI algorithm is continuously delivered to edge sensors.

This chapter aims to present some edge-to-cloud solutions on the market namely Azure IoT Edge, AWS Greengrass, Google Cloud Platform, Nuvla/NuvlaBox, and Balena. Nuvla/NuvlaBox is a solution developed by SixSq (a swiss company). A study will be done to look for the solutions adequate for our edge-based FPGA.

Figure 2.1 depicts a scenario that will allow us to describe and compare each solution. The SONAL application consumes data coming from the microphones. The AI models, which compose the SONAL application, deployed on the edge-device are responsible for making "predictions" based on the input sensing data in order to detect and classify the vehicles passing by the sensors. These AI models will be trained in the Cloud and the inference versions will be deployed on the edge.

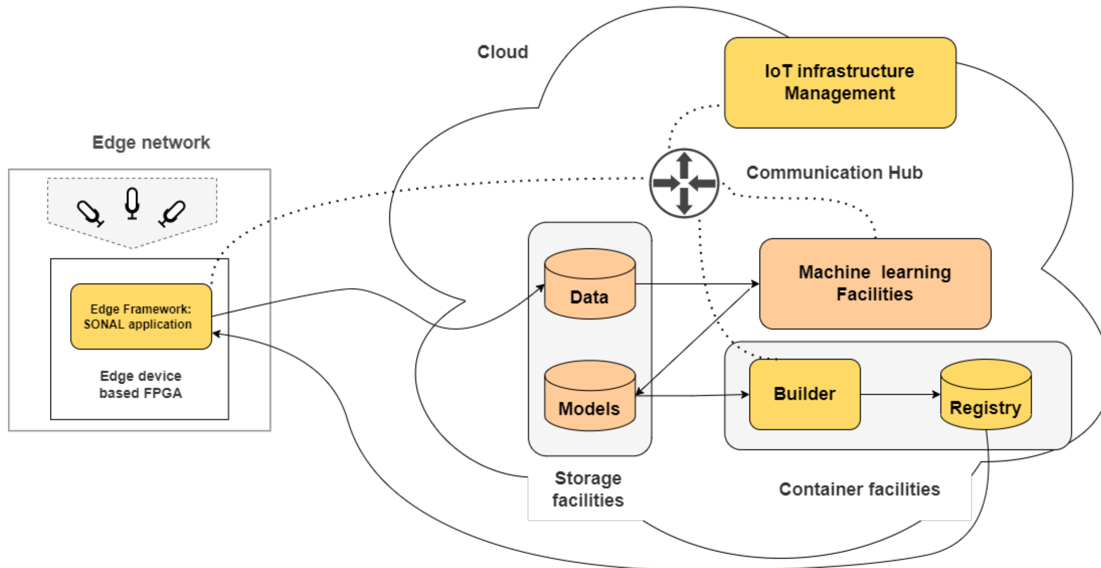

Figure 2.1: A self-adaptive SONAL application scenario

So in this scope, basic components are chosen to do the comparison study between those services. Those components are colored in Figure 2.1. The components specific to the learning phase are shown in orange, while the components, composing the edge to cloud infrastructure, are shown in yellow. The components are listed below :

- **IoT Infrastructure Management** Play the role of orchestration by provisioning (configure and deploy) and monitoring Edge/IoT networks. Edges devices are mostly autonomous. Indeed, the edge framework modules are packaged and provisioned to the edge via the Cloud.
- **Edge Framework** This is typically in the form Software Developmpent Kit (SDK) to facilitate the programming and execution of edge modules. Therefore, you can manage the edge runtime through an Application Programming Interface(API), as well as tools for controlling the edge such as a command-line interface (CLI).
- **Container Facilities** Represent the ability to containerize an application and save the image in a Cloud registry for deployment on edge devices. The containers would then be pulled directly from the registry by edge devices.

- **Communication Hub** Creates a message architecture for the edge modules to communicate with one another. One of the popular communication strategies is MQTT which stands for Message Queuing Telemetry Transport and is based on event transmission. It is a lightweight and battery-friendly communication technique. Other protocols, such as AMQP (Advanced Message Queuing Protocol), may be available. AMQP supports a wide range of messaging patterns and is very easy to extend. Furthermore, HTTPS is inefficient in comparison to the previous protocols.

- **Storage Facilities** Represent the ability to store relevant data streamed from the edge, in order to use it for training the AI models. It could be used also to store the newly performed version of AI models.

- **Feasibility** The edge-based FPGA supports a customised Linux image. In this case, the edge-to-cloud technology must be compatible with the hardware.

- **Machine learning Facilities** Builds and trains the intelligent part of SONAL application in the cloud. Human involvement is necessary to label audio frames. The Edge Framework notifies the Cloud, via the communication hub, of the availability of fresh low-performance data, allowing for the coordination of a learning round: the result is a new version of the vehicle classification model. In general, for this part, we will use instances in the cloud as our model is based on CNN to classify audios.

2.2 Balena

Balena is a cloud solution for IoT that enables building, deploying code to a fleet (a group) of connected devices, and managing them through the BalenaCloud dashboard.

- **IoT Infrastructure Management** Devices are tightly integrated into a Balena ecosystem via BalenaOS and supervised by BalenaCloud. Managing devices is proceeded through some tools such as variables, ssh access. It provides also remote log watching on its web dashboard where those messages could be sent from the device supervisor or written by the services to stdout and stderr.

- **Edge Framework:** Balena offers BalenaOS [7] which is yocto Linux-based OS. The host OS is responsible for launching the device supervisor as well as the containerised services. Balena has its own docker version BalenaEngine [5] which is a container runtime based on a lightweight version of Docker's Moby.

- **Container Facilities** BalenaCloud offers a container registry and a remote builder service. Balena CLI, which is installed on the development host, is used for deployment. "balena push" compresses the application and sends it to BalenaCloud builder, which builds the images in an environment that suits the devices' architecture. The images are saved in the balena registry, and the balena device supervisor is notified to deploy this container.

- **Communication Hub** Balena does not provide application messaging services. It is the responsibility of the application developer to select/implement one and ship it with an OS image update.

- **Storage Facilities** Balena does not provide Cloud storage for application data, though any S3-compatible solution can be used.

- **Feasibility** In order to integrate a device with BalenaCloud, the device must run the BalenaOS operating system. The packages of BalenaOS userspace contain only the minimum services required for deploying and running containers. Figure 2.2 illustrates the architecture of Balena infrastructure [6].

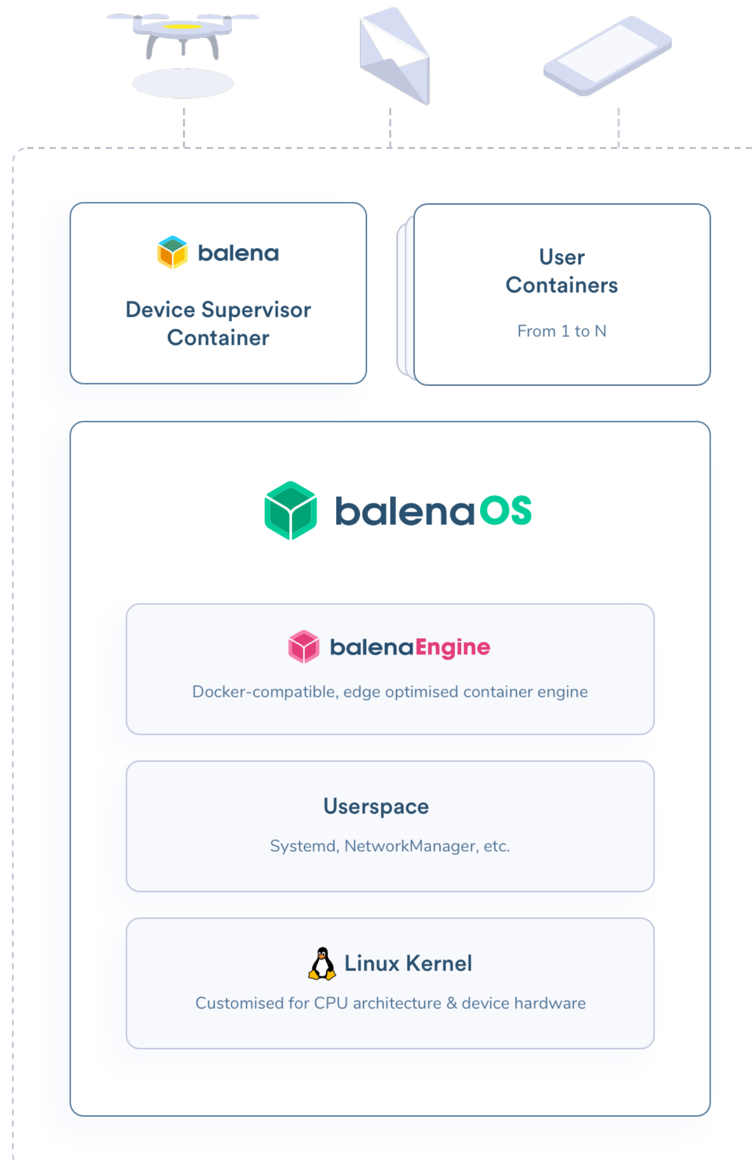

Figure 2.2: Balena architecture

This project's device necessitates the use of a customised operating system. The edge is indeed FPGA-based, and the hardware used has a specific architecture MPSoC multiprocessor on chip. As a result, a customised operating system is provided, that includes the necessary libraries and drivers to ensure communication between those processors.

2.3 Google Cloud Platform

Until now, Google Cloud Platform (GCP) has provided a basic IoT connectivity Cloud IoT Core that allows remote control of devices and data collection from IoT sensors. Google provides the Coral board, which includes an integrated Tensor Processing Unit (TPU), allowing for faster AI inference.

- **IoT Infrastructure Management** Cloud IoT Core [12] is made up of two modules. Device Manager allows you to remotely authenticate, configure, and control IoT sensors. Whereas the Protocol Bridge communicates using the MQTT and HTTP protocols. The data is streamed to the Cloud via Pub/Sub service. There is no developed orchestration service.
- **Edge Framework** Google, unlike AWS, Microsoft Azure, and SixSq Nuvla, does not offer a packed Edge Framework component.
- **Container Facilities** Google Container Registry [12] is a service that allows you to save Docker images privately.
- **Communication Hub** Edge devices can leverage the MQTT or HTTP protocol "bridge" provided by the Pub/Sub [19] service to publish telemetry events, obtain configuration data, and set device state. As a result, messages can be utilised to trigger additional Cloud processing, such as analytics and machine learning (re)training.
- **Storage Facilities** Cloud Storage [8]. A RESTful object storage Web service with a proprietary interface.
- **Feasibility** Not available.

2.4 Azure IoT Edge

Azure IoT Edge is a service provided by Azure to remotely manage services deployed and executed locally on IoT Edge-enabled devices.

- **IoT Infrastructure Management** IoT Hu [15] is a fully PAAS managed service. This last is responsible for registering devices, authenticating per device for security-enhanced, managing IoT devices at scale, and above all monitoring user applications deployed on the edge. As well, it offers device-to-cloud and cloud-to-device communication options (MQTT or AMQP).
- **Edge Framework** In this case, we are talking about IoT Edge Runtime [18] which has the responsibility to manage the modules deployed to the edge device. Edge Hub and Edge Agent are the two runtime modules that make up this system. The Edge Hub acts as a proxy that ensures the communication between the IoT Infrastructure Management (IoT Hub) and the Edge-enabled device. The Edge Agent is in charge of deploying and managing the user modules which represent the edge application code.
- **Container Facilities** Azure Container Registry [4] is a private registry based on the open source Docker Registry 2.0. We can use any other publicly accessible Docker registry.
- **Communication Hub** Azure provides IoT Hub message routing services [16] in order to ensure the communication between user modules deployed on the edge or communication between the edge and Azure Cloud. Event Hubs or the Event Grid service can be used to further process telemetry and application messages in the Cloud.
- **Storage Facilities** Azure Storage [3] covers various data types such as blobs, files, queues, tables, and disks. The storage account offers a unique namespace to access data from anywhere over HTTP or HTTPS (RESTful).
- **Feasibilty** Azure IoT Edge is a Docker container orchestration platform. To prepare the Azure IoT Edge runtime, we must have docker moby installed on the device.

2.5 NuvlaEdge

The Nuvla solution is also concerned with fleet management and containerized application deployment.

- **IoT Infrastructure Management** Nuvla is available as either a stand-alone software stack for installation on-premises or as a PaaS via Nuvla.io [22]. The user application is packaged as Docker images, and saved in the registry. A docker-compose is used to create the application in Nuvla cloud. Nuvla provides an interface App store to deploy the user application to the edge devices.

- **Edge Framework** NuvlaEdge [23] runtime software. It is a set of microservices managed by Nuvla cloud service in order to transform any device into an Edge device. They support VPN-based networking with Nuvla and within the same edge device, MQTT-based internal messaging, application monitoring, security, and discovery of attached HW components like network devices, GPU boards, etc.

- **Container Facilities** Nuvla supports Docker with any public or private registry.

- **Communication Hub** An internal MQTT messaging system is provided by Nuvla to ensure the communication between NuvlaEdge components and any user container running in the same internal network. For upstream communication with Nuvla.io, an HTTP-based RESTful API is used as well as a Python client. Nuvla does not offer a private Communication Hub service, but its private App store makes it simple to set one up.

- **Storage Facilities** We can use any third-party S3. Nuvla offers an S3 metadata catalog.

- **Feasibility** Nuvla is a Docker container orchestration platform. In order to prepare the Nuvla edge runtime, we must have "docker" and "docker-compose" installed on the device.

2.6 AWS GreenGrass

Amazon Web Services (AWS) provides a solution to build, deploy and manage services on edge devices. These services can be serverless with lambda technology or containerised.

- **IoT Infrastructure Management** AWS offers AWS IoT Device Management [10] which is an agnostic service that allows you to manage, monitor, and track IoT devices. Greengrass core devices can be managed as a group by AWS Greengrass. Deployments can be arranged into device hierarchies; OTA updates and auditing are supported; and Amazon CloudWatch monitoring is available.
- **Edge Framework** AWS IoT Greengrass [14], which allows the device to connect to AWS services and third-party apps, also allows the device to run AWS Lamda functions or Docker containers locally.
- **Container Facilities** AWS offers Amazon Elastic Container Registry (ECR) [20] which is a private Docker registry.
- **Communication Hub** AWS IoT Core, a message broker based on the MQTT protocol, is available from AWS. It is a secure service that uses TLS 1.2 encryption and is available across the entire AWS ecosystem, from edge services to cloud storage facilities. It can also handle disconnected devices. The EventBridge service handles additional processing.
- **Storage Facilities** AWS provides Object Storage Service well suited for raw sensing data and AI definition files. The service is intended for online backup and archiving of data on Amazon.
- **Feasibility** AWS IoT Greengrass officially supports devices with the following architectures: Armv7l, Armv8 (AArch64), x86_64 The device architecture in our project is AArch64, so the solution is suitable for the hardware we are using.

2.7 Conclusion

This comparative study reveals that the overall architecture of edge-to-cloud solutions is quite similar. Almost all of them are based on Docker except Balena which provides a whole ecosystem, starting from the operating system, in order to monitor a device. Therefore, we chose Azure IoT Edge and Nuvla/NuvlaEdge solutions to test the deployment of the SONAL application on the edge-based FPGA.

CHAPTER 3

FPGA ACCELERATED APPLICATION

3.1 Introduction

After conducting research into edge-to-cloud solutions that would be appropriate for our targeting solution, let's delve into the specifics of our work's hardware and software environment. As a result, this chapter aims to present the distinctive features of the FPGA and the implementation of SONAL software on FPGA infrastructure.

This chapter is composed of three parts: The first section explores the architectures of FPGA, as well as the advantages of using these devices. The Edge device hardware and operating system are described in the second section. The third section describes the structure of an application interfacing with an FPGA by presenting a sample application "vector-add" which computes the addition of two vectors using FPGA. The fourth and final section presents the DPU accelerator for Convolutional Neural Network (CNN) application.

3.2 Field Programmable Gate Arrays

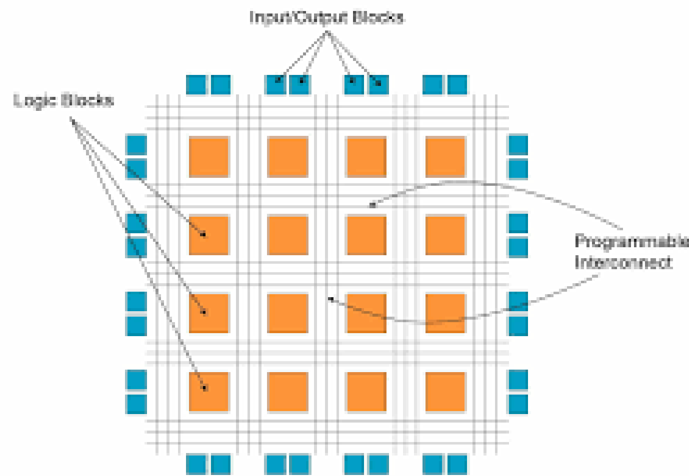

Figure 3.1: The basic structure of the FPGA

An FPGA is a semiconductor device, made up of a collection of "logic blocks", "Input/Output Cells" and "interconnection resources", as shown in figure 3.1, that can be reconfigured to connect the inputs and outputs (I/O) and logic blocks in a variety of ways to perform massively paralleled, real-time processing. This specific architecture, targeting the particular computation requirements of the algorithm, results in a more optimal computing hardware. As a result, FPGAs devices represent a promising solution for low power computation.

CPUs and GPUs are instruction-based architectures, this makes them easier to connect via software-based methods. This makes it easier for hackers to update and adjust the system. However, FPGAs are configured by specifying a hardware circuit where the uniqueness of the task-driven FPGA requires unique tools to create, special skills to program. This means that not only we have more control over our hardware by design, but also it's much more difficult to perform side channel attacks or reverse-engineering on FPGA.

FPGAs are configured via a bitstream which is represented by a file containing the binary data that codes the chip. One of the most significant advantages of FPGAs is the ability to perform encryption and authentication. Encryption helps preserve bitstream's confidentiality. Authentication ensures the integrity of the data and the source that is transmitting it. It helps to improve communication security by eliminating falsified information.

One of the major manufacturers of the FPGA is Xilinx [27], an industrial company, which engages in the designs and development of programmable logic semiconductor devices and related software design tools.

Xilinx provides **S**ystem **o**n **C**hip (SoC) FPGA device which is an integration of a standalone processor with the FPGA architecture. Having the processor and the FPGA fabric on the same silicon chip tremendously reduces production costs and saves space on the circuit board.

3.3 Zynq MPSoC hardware overview

This section introduces the edge-based FPGA platform, which is the Zynq UltraScale + MPSoC ZCU104 evaluation board from Xilinx, represented in Figure 3.2.

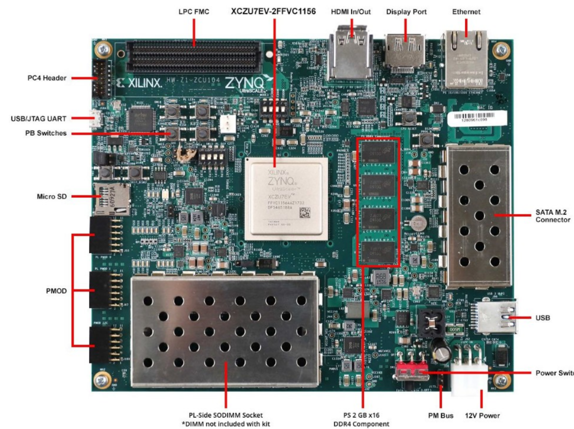

Figure 3.2: Zynq UltraScale + MPSoC ZCU104 evaluation board

As specified in Figure 3.2, XCZU7EV Multi-Processor System-On-Chip (MP-SOC) represents the heart of our platform which combines the processing system (PS) and the programmable logic (PL). The PS part can be connected to the PL part via multiple interfaces in order to integrate the hardware accelerators and other functions in the PL logic that are accessible to the processors. The processing system (PS is composed of three processing elements (Figure 3.3):

- Cortex-A53 Application Processing Unit (APU)-Arm v8 architecture-based 64-bit quad-core multiprocessing CPU.
- Cortex-R5 Real-time Processing Unit (RPU)-Arm v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM).
- Mali-400 Graphics Processing Unit (GPU)-graphics processing unit with pixel and geometry processor and 64 KB L2 cache.

PL part represents the Field programmable Gate Arrays(FPGA).

This platform's heterogeneity and various levels of parallelism make it a useful tool for a variety of applications such as signal processing, image processing, etc.

Both APU and FPGA can communicate together with the main memory via high-performance buses. The FPGA uses these busses to read/write an array of data from/to the main memory.

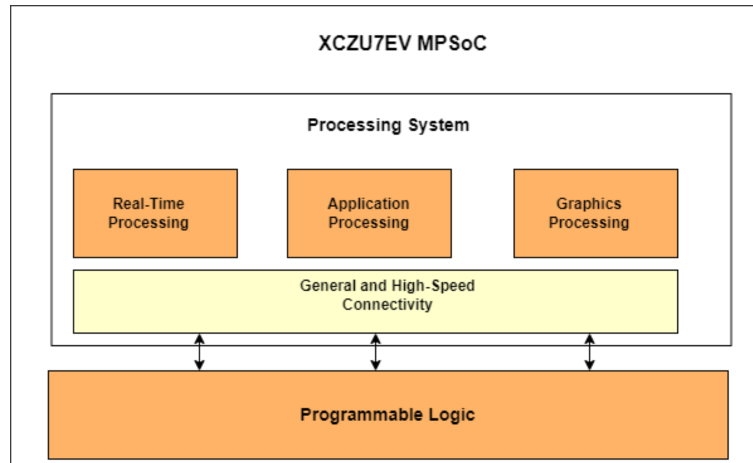

Figure 3.3: The base components of XCZU7EV MPSoC

As any embedded system, ZCU104 needs an operating system to work with. This last can be generated using some powerful tools such as petalinux or using a ready-customised ubuntu image provided recently on the 14th of December 2021 by Canonical and Xilinx [13].
This image is customised for Xilinx devices. All the necessary drivers and libraries needed, in order to communicate, use, and update the FPGA device, are provided in the Ubuntu image.

3.4 Vitis

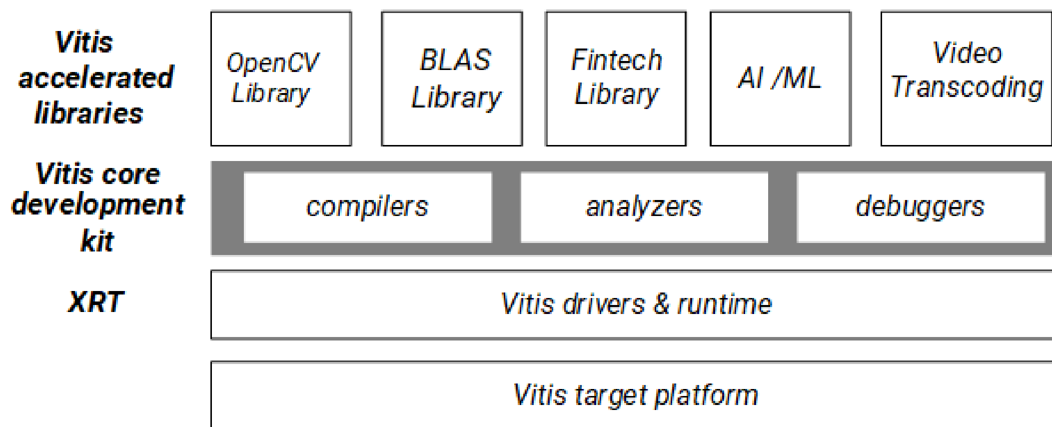

Figure 3.4: Vitis Unified Software Platform

Among the tools provided by Xilinx, there is Vitis which is an unified development environment for Xilinx devices. This tool allows the development of the software and hardware parts using languages adapted to software engineers and artificial intelligence, like C, C++ or Python, in order to take advantage of the FPGA. In other words, Vitis makes FPGA programming much more a question of software development rather than hardware design. Vitis Unified Software Platform, as shown in the Figure 3.4, consists of the following elements:

- **Vitis target platform** The target platform is the native hardware design for the FPGA before adding any accelerator or any custom logic.
- **Xilinx RunTime (XRT)** XRT provides the drivers and an API to connect the PS part with PL part. Indeed, it handles communication between the host application (running on the APU) and the accelerated kernels. Key functions of the Xilinx Runtime include: Downloading the FPGA binary / Memory management between host and accelerator / Board management.
- **Vitis core development kit** The Vitis core development kit, is a set of graphical and command-line developer tools that includes compilers and cross-compilers for compiling source code into executables that can run on a specific target device, as well as analyzers and debuggers for analysing application performance and locating problems.
- **Vitis accelerated libraries** Vitis accelerated libraries are a set of open-source, performance-optimised libraries that enable ready-to-use acceleration for your existing application (written in C, C++, or Python). Vitis libraries are provided for common arithmetic, statistics, linear algebra, and DSP functions, as well as domain-specific applications like image processing and data analytics. Vitis contains 26 libraries that cover a wide range of topics, including artificial intelligence, image processing, finance, security, and mathematics.

3.5 Accelerated application structure

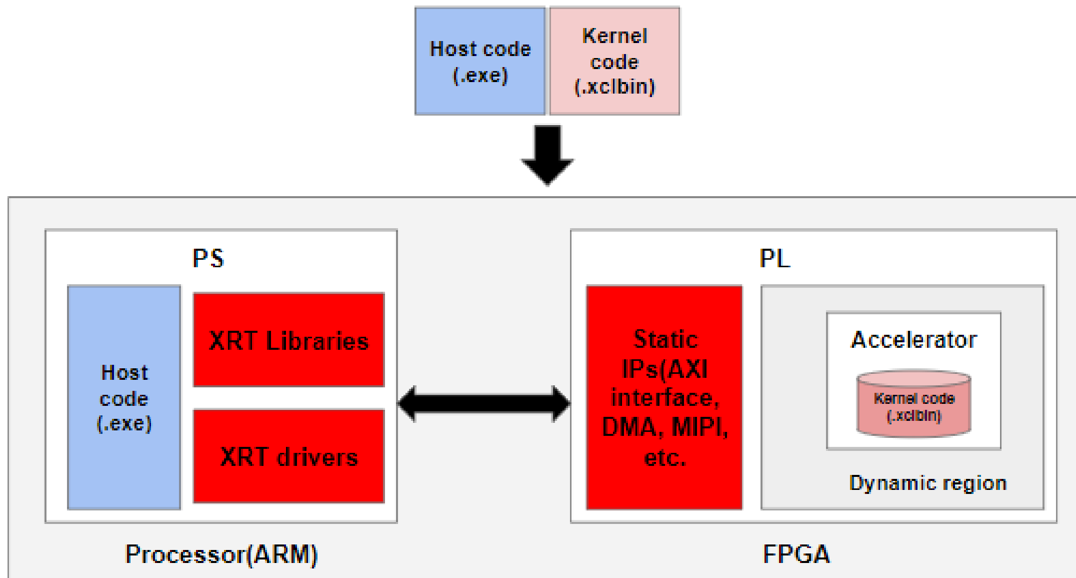

Figure 3.5: Structure of an application interacting with the FPGA device

The application, interfacing with the FPGA, is composed of two parts (Figure 3.5): host code and kernel code with communication channels between them.

- **Kernel code** is part of the application where the computational task to be executed is expressed. The kernel code is compiled into an executable device binary(.xclbin) on the FPGA.
- **Host code** runs on the APU and computes functions that run on the FPGA devices. Indeed, it is responsible for initialising the accelerator, allocating memory and setting up data transfer points between the host and device and triggering the execution of the kernels on the FPGA. The host program could be written in C/C++ , Python.

The transactions between the host program and the kernel, including control and data transfer, are managed by Xilinx Runtime (XRT) via API calls. The process occurs across an AXI (Advanced eXtensible Interface) bus.

3.6 Sample FPGA application

3.6.1 Prepare the kernel code and host code for Vector add

In this section, we create a trivial vector-add, a simple example that allows us to focus on the key concepts of FPGA acceleration on MPSoC. We used Vitis IDE to create the host code and kernel code for our application.

The objective of the vector-add function is to read two inputs vectors in1 and in2, and computes the sum.
The program, that will be loaded into the Programmable Logic part, is the kernel code discussed in the previous section. In this program, we described the architecture of the accelerator. Indeed, the accelerator reads through the AXI interfaces the 2 inputs in1 and in2, computes the sum operation which will be described in the host application, then map the output into another AXI interface.

The host code is programmed in C++, it manages the communication with the FPGA through a specific API:

1. Through the AXI interface, the host application writes the data required by a kernel into the global memory.
2. The host code sets up the kernel with its input parameters.
3. The host code triggers the execution of the kernel function on the FPGA.
4. The kernel code performs the required computation while reading data from global memory, as necessary.
5. The kernel code writes data back to global memory and notifies the host that it has completed its task.
6. The host code reads data back from global memory into the host memory and continues processing as needed.

To develop an accelerator that is compatible with our platform Zynq Ultrascale + MPSoC ZCU104. Vitis needs a hardware description of the device we are using. As shown in Figure 3.6, we have as input a hardware description file that describes the device's existing hardware. Vitis requires also the sysroot of the operating system to use its libraries and ensures the host code will work also within this operating system.

Vitis go through several stages: First, it constructs the accelerator. Then it connects this custom accelerator to the default platform. It is responsible for compiling the host code. In the end, we have an executable file that represents the host code, as well as a kernel code compatible with the hardware architecture of the platform Zynq Ultrascale + MPSoC ZCU104.
The Figure 3.6 resumes the input and the output of Vitis.

Figure 3.6: Compile the Vector add on Vitis

3.6.2 Prepare the Platform Assets Container

A Platform Assets Container (PAC) is used to package custom boot assets. Indeed, if we want to add an accelerator to the PL part, the system needs to be aware of it. The PAC is a folder with a specific hierarchy, it contains files that will be used in the boot process and kernel code for the accelerator which will be loaded in the PL part. Indeed, the artifacts in Figure 3.6 are needed:

The required files for the PAC are described in the Table 3.1.

3.6.3 Activate the PAC on the device

To use the accelerator, a command-line tool called **xlnx-config** [26] is used to manage and manipulate the hardware platform for Xilinx devices with Ubuntu operating system. The major goal of **xlnx-config** is to load custom hardware platforms, and it does so by managing the installation of custom boot assets (bitstream, firmware, and xclbin) while keeping the "golden" boot components that come with the certified Ubuntu image.

Once the PAC is added to the boot partition of the SD card we can use **xlnx-config -q** to query the system and report the boot assets available for the board. In Figure 3.7, we have the example of “zcu104_vitis_hello_world” which includes

Filename	Description
bootgen.bif	Bootgen config file used by xlnx-config to package new boot.bin
fsbl.elf	First Stage Boot Loader
bl31.elf	ARM Trusted firmware
pmufw.elf	Platfrom Management Unit Firmware
system.bit	The Programmable Logic bitstream
system.dtb	The Linux Device tree. This describes the hardware that can be read by an operating system
.xclbin	The container binary file specific to the accelerator. It will be written in PL part

Table 3.1: The required files in PAC

the assets related to the vector add application (host code and kernel code) and the boot assets for “Zynq UltraScale MPSoC ZCU104”.

```
ubuntu@zynqmp:~$ xlnx-config -q


PAC configurations present in the system:

| PAC Cfg                        |Act| zcu104 Assets Directory
---------------------------------------------------------------------------------------------------
| zcu104_vitis_hello_world       |   | /boot/firmware/xlnx-config/zcu104_vitis_hello_world/hwconfig/hello_world/zcu104
| zcu104_vitis_vehicle_detection|   | /boot/firmware/xlnx-config/zcu104_vitis_vehicle_detection/hwconfig/vehicle_detection/zcu104
---------------------------------------------------------------------------------------------------

* No configuration is currently activated *
```

Figure 3.7: Checking the current available HW configurations

xlnx -config -a in Figure 3.8 is used to activate the configuration by using the path to the manifest. It checks the system compatibility and then applies the configuration by regenerating a boot binary based on the new system that includes the new customised hardware.

```
ubuntu@zynqmp:~$ xlnx-config -q

PAC configurations present in the system:

| PAC Cfg                         |Act| zcu104 Assets Directory
--------------------------------------------------------------------------------------------------------
| zcu104_vitis_hello_world        |   | /boot/firmware/xlnx-config/zcu104_vitis_hello_world/hwconfig/hello_world/zcu104
| zcu104_vitis_vehicle_detection|   | /boot/firmware/xlnx-config/zcu104_vitis_vehicle_detection/hwconfig/vehicle_detection/zcu104
--------------------------------------------------------------------------------------------------------

* No configuration is currently activated *
```

Figure 3.8: Activate the configuration for the Vadd accelerator

The system must be restarted. **xlnx-config** will save the configuration after activation. When we run the command **xlnx-config -q** we can see that the first PAC has a star as shown in Figure 3.9. This indicates that the PAC containing the accelerator for the Vector add application is activated.

```
ubuntu@zynqmp:~$ xlnx-config -q

PAC configurations present in the system:

| PAC Cfg                         |Act| zcu104 Assets Directory
--------------------------------------------------------------------------------------------------------
| zcu104_vitis_hello_world        | * | /boot/firmware/xlnx-config/zcu104_vitis_hello_world/hwconfig/hello_world/zcu104
| zcu104_vitis_vehicle_detection|   | /boot/firmware/xlnx-config/zcu104_vitis_vehicle_detection/hwconfig/vehicle_detection/zcu104
--------------------------------------------------------------------------------------------------------
```

Figure 3.9: Checking the activated HW configurations

After configuring our platform with the appropriate accelerator, we can start our application by calling the host code (.exe) and passing as arguments the two inputs in1 and in2 as well as the kernel code (.xclbin). Figure 3.10 shows the execution of Vector add application. We display both the provided values to FPGA devices and the computation result.

```
ubuntu@zynqmp:/boot/firmware/zcu104_vitis_hello_world/hwconfig/hello_world/zcu104$ ./vaddd 1222 1111  binary_container_1.xclbin
INFO: Reading binary_container_1.xclbin
Loading: 'binary_container_1.xclbin'
Value 1 : 1222
Value 2 : 1111
Result : 2333
TEST PASSED
```

Figure 3.10: Executing Vector add application on the ZCU104 device

After running the application, we can notice in the Figure 3.11, the kernel code is loaded in the PL part.

```
ubuntu@zynqmp:~/src/test_app$ xbutil query
INFO: Found total 1 card(s), 1 are usable
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
System Configuration
OS name:        Linux
Release:        5.4.0-1015-xilinx-zynqmp
Version:        #18-Ubuntu SMP Tue Jul 13 07:06:07 UTC 2021
Machine:        aarch64
Glibc:          2.31
Distribution:   Ubuntu 20.04.2 LTS
Now:            Tue May 31 11:07:11 2022 GMT
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XRT Information
Version:        2.8.0
Git Hash:
Git Branch:
Build Date:     2021-05-28 11:59:54
ZOCL:           2.8.0,
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shell                           FPGA                            IDCode
edge                            N/A                             N/A
Vendor          Device          SubDevice       SubVendor
0x10ee          N/A             N/A             N/A
DDR size        DDR count       Clock0          Clock1          Clock2
4 GB            1               100             0               0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Memory Status
     Tag         Type        Temp(C)  Size    Mem Usage       BO count
[ 0] HPC0        **UNUSED**           2 GB    0 Byte          0
[ 1] HP3         **UNUSED**           2 GB    0 Byte          0
[ 2] HPC1        **UNUSED**           0 Byte  0 Byte          0
[ 3] HP0         MEM_DRAM             2 GB    0 Byte          0
[ 4] HP1         **UNUSED**           0 Byte  0 Byte          0
[ 5] HP2         **UNUSED**           0 Byte  0 Byte          0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Streams
     Tag         Flow ID  Route ID Status   Total (B/#)     Pending (B/#)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Xclbin UUID
1c78b56e-769f-ef32-e4b8-834001c07a23
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Compute Unit Status
CU[ 0]: krnl_vadd:krnl_vadd_1          @0x80000000        (IDLE)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
INFO: xbutil query succeeded.
```

Figure 3.11: Loading kernel code to PL part

3.7 Accelerate convolutional neural network

Unlike previous hardware designs focusing on specific functions, Vitis, specifically Vitis AI, supports a DPU (deep-learning processor unit), dedicated to the convolutional neural network (CNN). It also supports the basic functions of deep learning, and developers can take advantage of DPUs to accelerate CNN inference.

The DPU supports the following network features: Convolution, Depthwise convolution, Deconvolution, Max or Average pooling, and Full connexion, ReLU family(ReLU, ReLU6, and LeakyReLU) activations functions, Normalisation, and Split. It is integrated into the programmable logic (FPGA) of the Xilinx Zynq UltraScale+ MPSoC zcu104 and integrated into the processing system (ARM) through AXI interconnect to perform CNN inference.

Each DPU is defined by a fingerprint, which is a way to encode the configuration of the DPU.
So this allows the Vitis compiler to know how to compile the model for the right DPU, and during runtime, this allows to check that the compiled model and DPU are compatible.

In our project, the vehicle detection program is deployed on the FPGA using the DPU. We initialized the hardware platform with the right DPU. And we compiled the vehicle detection model to match the configuration of the DPU.

3.8 Conclusion

In this chapter, we went over basic terminology to help clarify the important notion of application acceleration on an FPGA. As a result, we may conclude that hardware-accelerated applications can be containerized and deployed via an Edge-to-cloud solution. As a reminder, our goal in this project is to be able to update the SONAL program via the cloud. For FFT and matrix multiplication presented in the first chapter, there is no need to change those parts. However, for the CNN, we can update the DPU from the cloud, by feeding it with the new version of the model which will configure the DPU on the programmable logic part. The next chapter will focus on containerizing the hardware-accelerated application.

CHAPTER 4

DEPLOYMENT OF FPGA-ACCELERATED APPLICATION

4.1 Introduction

Containerizing FPGA-accelerated applications have a variety of advantages, including simplicity of deployment, configuring the FPGAs remotely, device separation, and many more. Docker containers are most typically used to quickly deploy CPU-based applications across several machines. Thus, for hardware-accelerated applications, it is a challenging task.

After grasping the main notion of communication between the processing system and the programmable logic, and having a clear understanding of some basic terminologies linked to FPGA accelerated applications, this chapter will take us through the process of containerization.

This chapter is divided into four parts: The first section deals with containerizing FPGA-accelerated applications, while the second and third sections deal with delivering them via Azure IoT Edge and Nuvla, respectively. The last section presents the deployment of FFT part through Azure and Nuvla.

4.2 Containerize FPGA application

Before we go any further, it's essential to recognise the notion of containerization. A container is a software unit that encapsulates code and all of its dependencies. The benefits of containerization are numerous [17]. On the one hand, it enables the application to move from one computing environment to another quickly and reliably. Indeed, rather than creating the application for thousands of devices, we can just create a packaged version and distribute it to all devices, because the container bundles all of the dependencies. Container portability comes from their ability to share the host machine's operating system kernel. To put it another way, numerous

containers can run on the same machine and share the OS kernel, each running as isolated in user space.

Containerization, on the other hand, adds an additional layer of protection through its isolation. Because containers are decoupled from one another, each program runs in its self-contained environment. This ensures that even if one container's security is compromised, the security of the other containers on the same host is maintained. We chose Docker since it is the most widely used containerization tool [24].

The primary goal of Containerization in our project is to package the SONAL application and deploy it using a Cloud to the Edge FPGA. So, in the long run, we can update our program whenever a new version of the AI model with improved performance is released.

In this context, to illustrate the creation of the Docker image for the FPGA-accelerated application, we will use the Vector add example presented in the previous chapter.

As illustrated in Figure 4.1, to run the application inside the container we need to package anything that resides in the user space, especially the XRT libraries. As defined in the fourth section of chapter three, it is responsible for downloading the FPGA bitstream and managing the memory between the host and the accelerator. We still have a dependency on the kernel and the hardware. For kernel, the container shares it with the host application so it will have access to the drivers on kernel space.

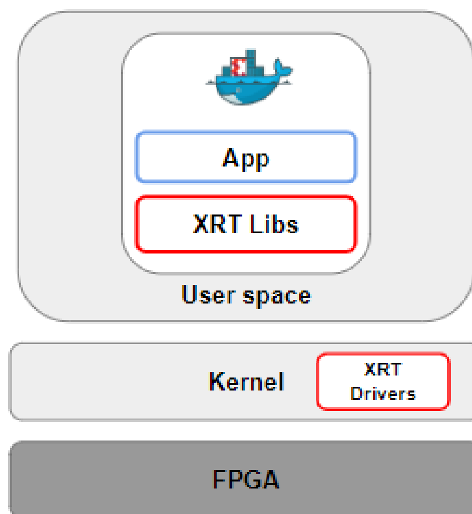

Figure 4.1: The environment needed for the container in order to communicate with the FPGA device

The first step is to develop a base docker image. A base image is an image that is used to create all of our container images. In this project, we create our own base image from scratch, based on the root filesystem of the Ubuntu Xilinx official image.

This is not the best solution because the image is 3.57GB in size, but it was the only option because the Xilinx official site does not have XRT packages for the embedded platform [28]

If this is the case, we can use Ubuntu as a base image, which is a lightweight image, and then install the necessary dependencies.

A base image stored on Docker Hub [9]. After that, we will use this image as a baseline for our containers.

Figure 4.2 represents the dockerfile to build the container. We need to copy the kernel code and the host code to the container.

```
FROM abir45ch/docker_base_img_xilinx
COPY ./binary_container_1.xclbin .
COPY ./vaddd .
CMD ./vaddd 1202 1032 binary_container_1.xclbin
```

Figure 4.2: Pass the host code and the kernel code to the container

Docker containers are "unprivileged" by default. This is due to the fact that a container is not able to access any devices by default, but a "privileged" container is. The—**device** flag can be used to restrict access to a specific device or devices. It gives us the option of specifying one or more devices that will be accessible within the container.

In this example, we limited the access to FPGA device by mentioning **“–device=/dev/dri/rend** ”.

```
ubuntu@zynqmp:~/src/test_app$ docker run --rm  --device=/dev/dri/renderD128:/dev/dri/renderD128 vadd_fpga1
INFO: Reading binary_container_1.xclbin
Loading: 'binary_container_1.xclbin'
Value 1 : 1202
Value 2 : 1032
Result : 2234
TEST PASSED
```

Figure 4.3: Run the Vector add container

The two vectors are added using the FPGA device after successful execution (Figure 4.3), and the result is displayed through the container.

4.3 Deployment through Azure IoT Edge

As mentioned in the second chapter: Azure provides an environment for edge computing to monitor and install code on devices. To turn a device into an Azure edge device, we must first create a device identity for our IoT edge device so that it can communicate with the IoT hub, and then associate a physical device with a device identity using a unique device connection string. The state of the device linked to Azure IoT Edge Cloud is shown in figure4.4.

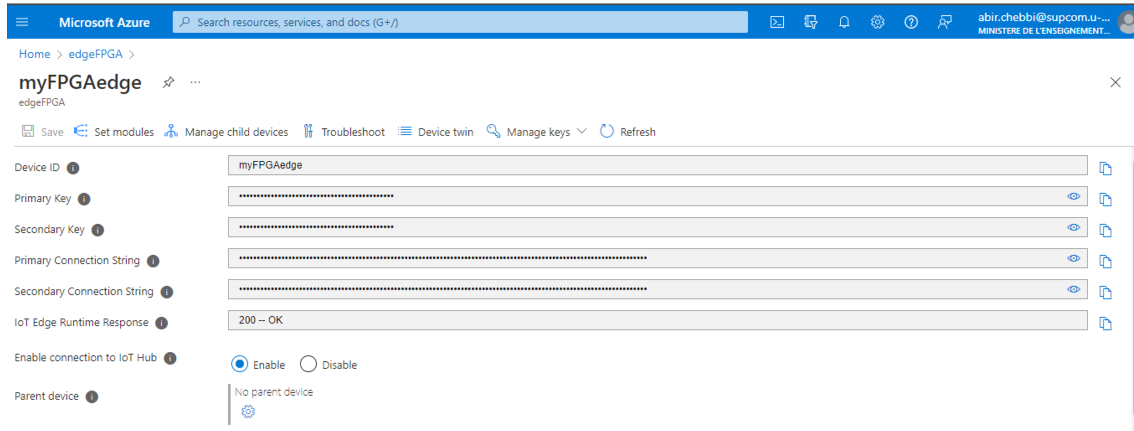

Figure 4.4: The Edge device based FPGA is connected to Azure IoT edge

When a device is connected to Azure IoT Edge, two docker containers, shown in figure 4.5, stand out: The IoT Edge agent, which makes it easier to deploy and monitor modules on IoT Edge devices, And the IoT hub, which is in charge of communication between modules on the IoT Edge device, as well as communication between the device and the IoT Hub.

Modules | IoT Edge hub connections | Deployments and Configurations

Name	Type	Specified in Deployment	Reported by Device	Runtime Status	Exit Code
$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes	running	0
$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes	running	0

Figure 4.5: Two docker containers running on the Edge device based FPGA and representing the IoT Edge Runtime

After connecting the device to the IoT Hub and configuring the Edge runtime on the device, we can now use the Cloud to deploy our Vector-add application. To accomplish so, we built and stored our image application using Azure Container Registry (Figure 4.6). In Azure, you can develop, store, and manage container images and artifacts in a private registry, which is a great feature.

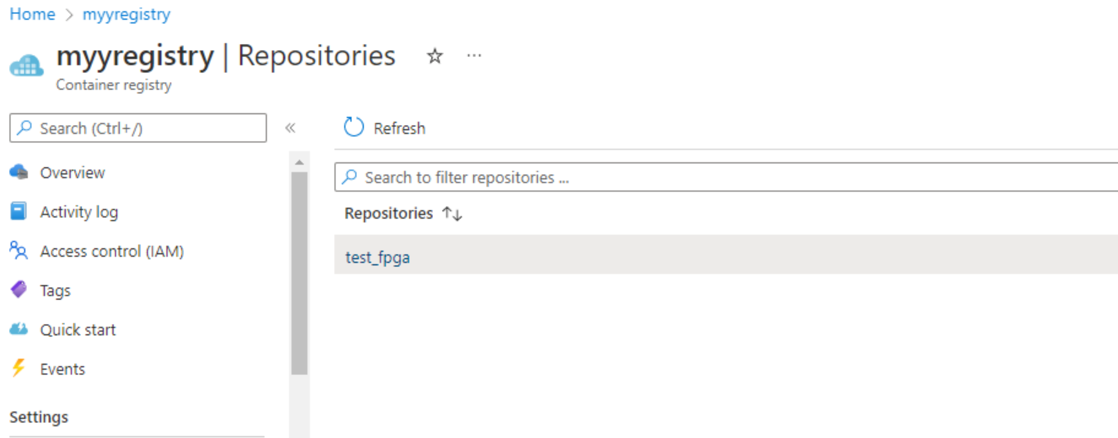

Figure 4.6: Vector add image stored in Azure Container Registry

Figure 4.7 illustrates the docker containers running on the Edge-based FPGA device. The two components for Edge Runtime and the FPGA accelerated application. To conclude, we create a Vector add image and store it on Azure Container

Modules | IoT Edge hub connections | Deployments and Configurations

Name	Type	Specified in Deployment	Reported by Device	Runtime Status	Exit Code
$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes	running	0
$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes	running	0
DefenderIotMicroAgent	Module Identity	NA	NA	NA	NA
test_fpga	IoT Edge Custom Module	✓ Yes	✓ Yes	running	0

Figure 4.7: Vector add module running on the Edge-based FPGA device

Registry, as seen in Figure 4.8. Then we distributed it to an FPGA device at the edge. After that, we can see the module successfully executing on the edge by submitting the operation to the FPGA. The outcome can be viewed using the Azure Interface (Figure 4.9).

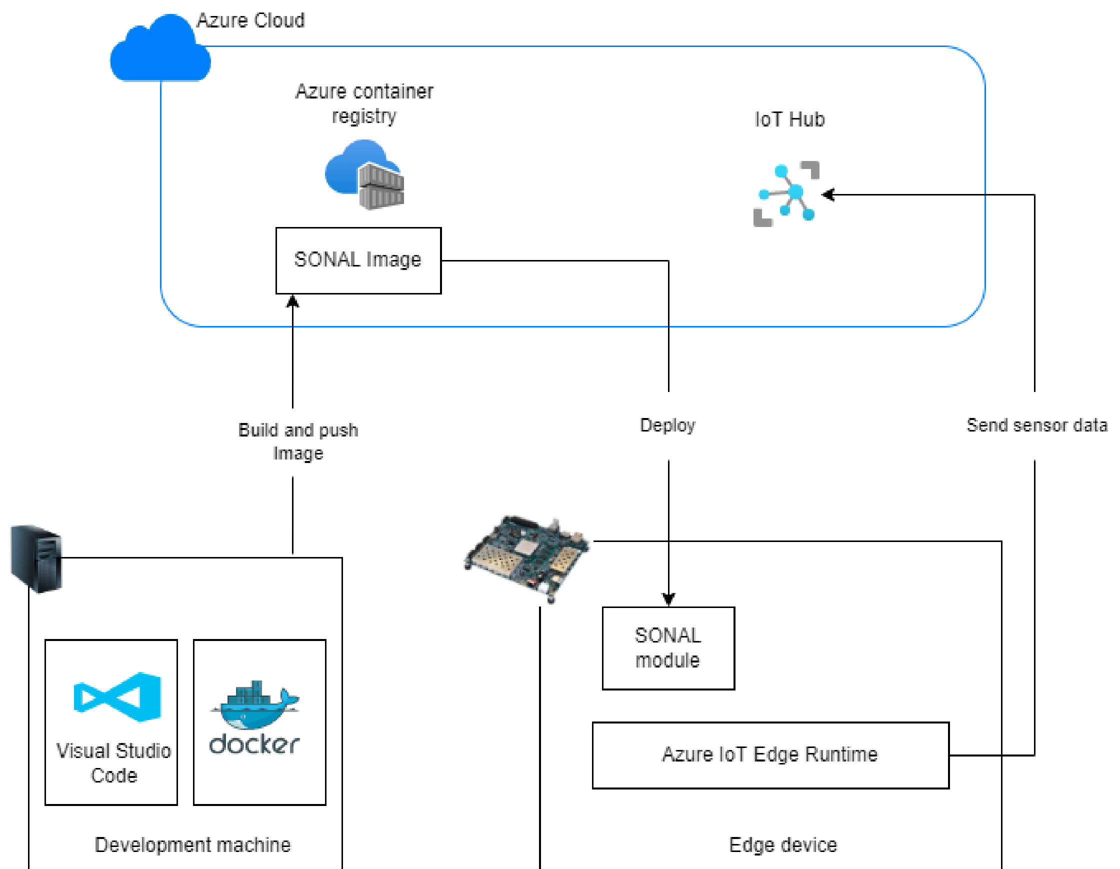

Figure 4.8: The workflow to deploy an application through Azure IoT Edge to the Edge-based FPGA device

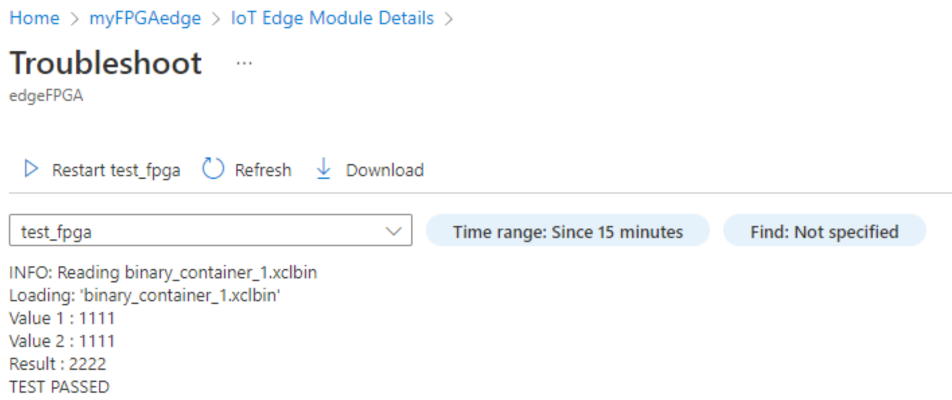

Figure 4.9: Vector add through Azure interface

4.4 Deployment through Nuvla/NuvlaBox

Nuvla provides an environment for edge computing to monitor and install code on devices, as presented in chapter two. To make the connection between the edge-based FPGA device and Nuvla possible. NuvlaEdge Engine must be installed in this one.

NuvlaEdge is launched through a docker-compose file that connects our edge device to the Nuvla. Once we run it, we have our edge device linked to the Nuvla cloud. After passing through those steps, we achieved to connect the edge-based FPGA to Nuvla.io, as shown in Figure 4.10. When a device is connected to Nuvla,

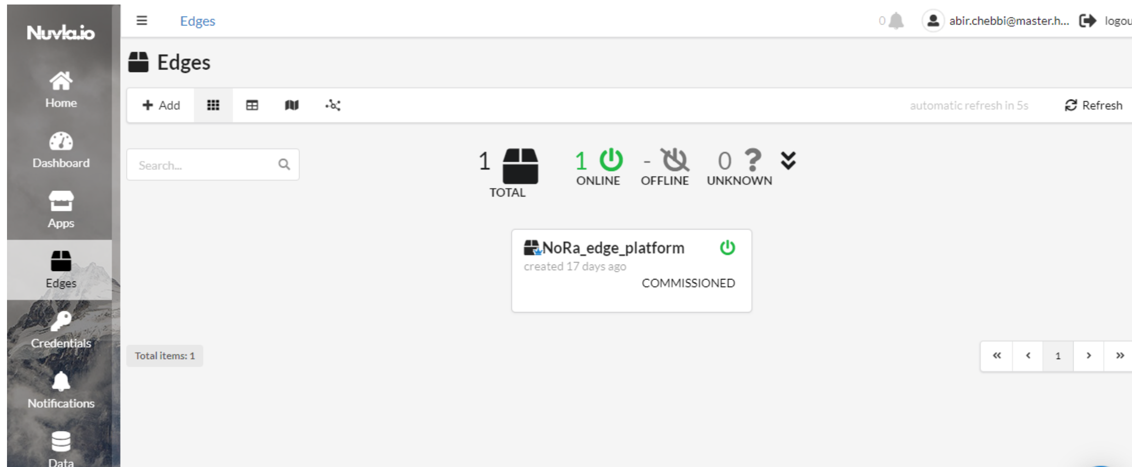

Figure 4.10: The Edge based FPGA device connected to Nuvla.io

several components, docker containers, stand out, which represent the runtime of NuvlaEdge:

- Agent:
  - Telemetry report
  - Communication with Nuvla
- System Manager:
  - Container supervisor
  - Data Gateway controller
- Compute-API:
  - Host Docker TCP connection
  - Key registration
- VPN Client:
  - A secure VPN tunne to ssh to the edge device
- Security:
  - Vulnerabilities report
  - Periodical security check
- Data Gateway:

  - Sending messages
  - Acquisition of sensor data

- Job Engine:
  - Job processing
  - Connectionless oriented reader

- On Stop:
  - Deployment cleanup

We may now launch the FPGA application once the device is attached. To do so, Nuvla.io offers a "App Store" interface via which you may develop the application as a docker-compose and then deploy it to the edge device, as illustrated in Figure 4.11. The device will pull the image from Docker Hub when you click deploy on Nuvla.io.

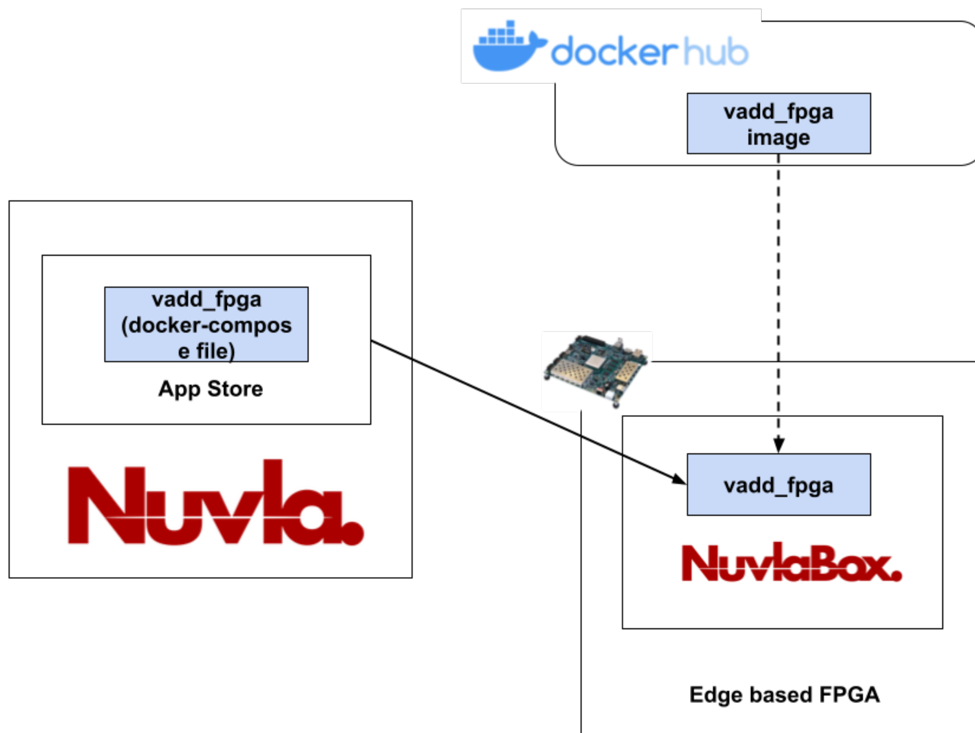

Figure 4.11: Deploy Vector add application on NuvlaBox

After deployment we can see through logs of the vadd_fpga container that is successfully computed on the FPGA as illustrated in Figure 4.12

```
ubuntu@zynqmp:~/src/test_app$ docker run --rm  --device=/dev/dri/renderD128:/dev/dri/renderD128 vadd_fpga1
INFO: Reading binary_container_1.xclbin
Loading: 'binary_container_1.xclbin'
Value 1 : 1202
Value 2 : 1032
Result : 2234
TEST PASSED
```

Figure 4.12: Vadd_fpga application running in NuvlaBox

4.5 Deploy FFT accelerated application

As the SONAL application is not yet ready to work on it. We tested deploying the FFT part. FFT's basic concept is to take a set of time domain waveform samples and process them to create a new set of frequency domain spectrum samples.

The same procedures had been taken with Vitis to construct the FFT accelerator. We have two outputs from Vitis, as we discussed in the vector add example. We have kernel code and instead of building a host code in the format ".exe", we generate a library that will be used in SONAL code.

Using numpy.rfft on the CPU, the total time to compute the FFT was roughly 52 milliseconds. Numpy.rfft is a function that uses an efficient algorithm called the Fast Fourier Transform (FFT) supplied by numpy to compute the one-dimensional n-point discrete Fourier Transform (DFT) of a real-valued array. For the same data, the execution time is roughly 16 ms.

To put it another way, the FFT is three times faster. In SONAL application the FFT is used continuously, so the edge device will gain more time to execute this part.

After ensuring that the FFT working locally, we containerized a part of the SONAL application that uses the FFT and we deployed it through Nuvla and Azure as demonstrated in Figure 4.13 and Figure 4.14

ID	Container Name	CPU %	Mem Usage/Limit	Mem %	Net I/O	Block I/O	Status	Restart Count
f3133997	FFT_FPGA	0.00	180.43MiB / 1.94GiB	9.08	0.01MB / 0.0MB	0.0MB / 0.0MB	running	0
55ade0cf	data-gateway.1.8fdrjrtcmv	0.07	2.79MiB / 1.94GiB	0.14	0.51MB / 0.09MB	0.0MB / 0.0MB	running	0
86ab7f0d	nuvlabox_security_1	0.00	17.1MiB / 1.94GiB	0.86	0.0MB / 0.0MB	0.0MB / 0.0MB	running	0
bfb7f63a	vpn-client	0.31	6.1MiB / 1.94GiB	0.31	0.0MB / 0.0MB	0.0MB / 0.0MB	running	0
68d476cf	nuvlabox_agent_1	0.81	51.76MiB / 1.94GiB	2.61	2.26MB / 1.63MB	0.0MB / 0.0MB	running	0
e88a0144	nuvlabox_system-manager_1	1.88	64.09MiB / 1.94GiB	3.23	0.28MB / 0.27MB	0.0MB / 0.0MB	running	0
e4a48d74	compute-api	0.00	3.73MiB / 1.94GiB	0.19	0.46MB / 1.3MB	0.0MB / 0.0MB	running	0
b0dd6d83	nuvlabox-on-stop	0.00	17.12MiB / 1.94GiB	0.86	0.01MB / 0.0MB	0.0MB / 0.0MB	paused	0
95a37e0f	nuvlabox-job-engine-lite	0.00	24.8MiB / 1.94GiB	1.25	0.01MB / 0.0MB	0.0MB / 0.0MB	paused	0

Figure 4.13: FFT container runing on NuvlaBox

```
ubuntu@zynqmp:~$ iotedge list
NAME            STATUS          DESCRIPTION     CONFIG
FFT_FPGA        running         Up 2 seconds    myyregistry.azurecr.io/fft_fpga:latest
edgeAgent       running         Up 23 seconds   mcr.microsoft.com/azureiotedge-agent:1.1
edgeHub         running         Up 0 seconds    mcr.microsoft.com/azureiotedge-hub:1.1
```

Figure 4.14: FFT container running on Azure IoT Edge

4.6 Conclusion

As we saw with the Vector add accelerator and FFT part, FPGA applications can be provided as Docker images. Using Docker to execute accelerated applications has various advantages over running them directly on a host server. It provides for a self-contained, pre-validated setup within a shareable image that can be readily distributed via Docker Hub or any other registry.

GENERAL CONCLUSION

As noise pollution becomes a major public health concern, the Noise Radar project could be a very useful solution in the Noise market. On the one hand, it provides a useful service and a great need for many companies, particularly governmental entities, essentially by letting go of the old method of manually controlling annoying vehicles with the policeman. On the other hand, this project may assist the government in creating a map of noisy areas.

The idea of the project is to develop a novel Noise Radar technology, capable of automatically identifying noise activities for different classes of vehicles via acoustic monitoring. As performance and power consumption constraints must be satisfied, while permitting system evolvability with a security-by-design approach, a solution based on an FPGA was chosen as the best trade-off. This represents a challenging task for us. Our work aims to implement an adaptive, Edge-Cloud-based FPGA NoRa platform, for sensors management at a city scale. The self-adaptive aspect provides automatic provisioning and continuous delivery of intelligence to the sensors.

To do so, we began by looking for edge-to-cloud solutions that would be suitable for the hardware we are developing. This research assisted us in highlighting the most widely used edge-to-cloud technologies. As a result of this step, we ruled out Balena because it did not fit our use case, as well as Google. For the rest of the solutions, Azure IoT Edge, Nuvla, and AWS are container-based technologies, that can be used in almost any hardware environment. We did not select the best technology for two reasons. First and foremost, this is dependent on the customer, so our goal is to make it open and provide a variety of solutions. Second, doing this type of comparison is dependent on the cost of the application, which we can't estimate right now because we need to have the SONAL application ready to estimate the type of instances that can be used in the cloud to train the models.

As a second step, we interfaced with the workflow of an accelerated FPGA application, starting with a sample application, to gain a thorough understanding of this type of application, as the work will be surely the same for the SONAL application.

Making the application work on Ubuntu was a bit difficult due to the recent release of this image for the Zynq UltraScale+ MPSoC ZCU104. When we were working on it, the documentation for activating the accelerators on Xilinx's official website was unavailable.

The following step was to containerize the FPGA-accelerated application. We worked to ensure that the same dependencies and library versions were present inside the container so that the application could communicate with the FPGA. This step enabled us to proceed and deploy the accelerated application via the cloud. As a result, we are confident that we can self-adapt the SONAL applications, which include some FPGA-accelerated applications.

During this work, we faced some difficulties at the beginning. Indeed, the use of the first board provided by Avnet [2] was a constraint to test these technologies because it supports a customised OS 'PetaLinux'. This OS is defined as a stack of layers. Each layer represents a software application that needs to be added/configured/recompiled at each software installation. This case doesn't align with the workflow of this project and will be a constraint in the future because PetaLinux does not support any package managers which help install software applications on the OS. Adding a software application requires layers, and rebuilding the OS. By using the Zynq UltraScale + MPSOC ZCU104 board which supports the Ubuntu distribution, we were able to test the different edge-to-cloud technologies without dealing with OS issues.

As future work, We will optimise the base docker image, which serves as the foundation for our containers, because it is large and takes time to deploy on the edge device.
Another aspect to consider is the security of the DPU portion that is used to speed up the classification and detection Neural Networks. The model loaded to the DPU, in this case, is not a bitstream. Thus, to protect Securaxis' intellectual property, we need to look for more security in this area.

REFERENCES

[1] ② *NORA: Noise Radar - Distributed Acoustic Sensor Network for Traffic Noise Impact Measurement - Données de base*. Dec. 2020. URL: https://www.aramis-a.admin.ch/Grunddaten/?ProjectID=47896.

[2] *AES-ZU3EG-1-SK-G*. June 2022. URL: https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-zu3eg-1-sk-g-3074457345635014225.

[3] *Azure Cloud Storage Solutions and Services | Microsoft Azure*. June 2022. URL: https://azure.microsoft.com/en-us/product-categories/storage.

[4] *Azure Container Registry | Microsoft Azure*. June 2022. URL: https://azure.microsoft.com/en-us/services/container-registry.

[5] *balenaEngine - A container engine purpose-built for IoT devices*. June 2022. URL: https://www.balena.io/engine.

[6] *balenaOS - Docs*. June 2022. URL: https://www.balena.io/os/docs/architecture.

[7] *balenaOS - Run Docker containers on embedded IoT devices*. June 2022. URL: https://www.balena.io/os.

[8] *Cloud Storage | Google Cloud*. May 2022. URL: https://cloud.google.com/storage.

[9] *Docker Hub*. June 2022. URL: https://hub.docker.com/repository/docker/abir45ch/docker_base_img_xilinx.

[10] *Gestion des appareils IoT | Intégration, organisation et mise à jour à distance | AWS IoT Device Management*. June 2022. URL: https://aws.amazon.com/fr/iot-device-management.

[11] *Global Noise Monitoring Market Size and Growth Forecast 2025*. June 2022. URL: https://www.bccresearch.com/partners/verified-market-research/global-noise-monitoring-market.html.

[12] *Google Cloud IoT – Services IoT entièrement gérés | Google Cloud*. June 2022. URL: https://cloud.google.com/solutions/iot.

[13] *Install Ubuntu on Xilinx | Ubuntu*. June 2022. URL: https://ubuntu.com/download/amd-xilinx.

[14] *Intelligence à la périphérie IoT – AWS IoT Greengrass – Amazon Web Services*. June 2022. URL: https://aws.amazon.com/fr/greengrass.

[15] *IoT Hub | Microsoft Azure*. June 2022. URL: https://azure.microsoft.com/en-us/services/iot-hub.

[16] kgremban. *Understand Azure IoT Hub messaging*. June 2022. URL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-messaging.

[17] Roberto Morabito et al. "Evaluating performance of containerized IoT services for clustered devices at the network edge". In: *IEEE Internet of Things Journal* 4.4 (2017), pp. 1019–1030.

[18] PatAltimore. *Learn how the runtime manages devices - Azure IoT Edge*. June 2022. URL: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-runtime?view=iotedge-2018-06.

[19] *Pub/Sub pour l'intégration des applications et des données | Cloud Pub/Sub | Google Cloud*. May 2022. URL: https://cloud.google.com/pubsub.

[20] *Registre de conteneurs entièrement géré – Tarification Amazon Elastic Container Registry – Amazon Web Services*. June 2022. URL: https://aws.amazon.com/fr/ecr.

[21] *Road traffic remains biggest source of noise pollution in Europe*. Nov. 2020. URL: https://www.eea.europa.eu/highlights/road-traffic-remains-biggest-source.

[22] SixSq S. A. *Enable your edge with our management platform as a service*. May 2022. URL: https://nuvla.io.

[23] SixSq S. A. *Secure and Intelligent Edge Computing Software*. June 2022. URL: https://sixsq.com/products-and-services/nuvlabox/overview.

[24] Gabriel N Schenker. *Learn Docker-Fundamentals of Docker 18. x: Everything you need to know about containerizing your applications and running them in production*. Packt Publishing Ltd, 2018.

[25] *Securaxis – sounds analytics for smart cities*. June 2022. URL: https://securaxis.com.

[26] *Snaps - xlnx-config Snap for Certified Ubuntu on Xilinx Devices - Xilinx Wiki - Confluence*. June 2022. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2057043969/Snaps+-+xlnx-config+Snap+for+Certified+Ubuntu+on+Xilinx+Devices.

[27] *Xilinx - Adaptable. Intelligent.* June 2022. URL: https://www.xilinx.com.

[28] *Xilinx Runtime Library (XRT)*. June 2022. URL: https://www.xilinx.com/products/design-tools/vitis/xrt.html#gettingstarted.