Kyushu Institute of Technology

Faculty of Engineering

PhD Thesis

Study on FPGA-based On-board Inferencing for Spacecrafts in Autonomous Operations

Cosmas Raymond Mutugi Kiruki

Embedded Systems Laboratory

Electrical and Space Systems

Supervisor:

Prof. Kenichi Asami

Thesis Committee:

Prof. Mengu Cho, Prof. Toyoda Kazuhiro, Prof. Takeshi Ikenaga
Department of Engineering

March 2021

Declaration

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of PhD, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

Kiruki Cosmas Raymond Mutugi
March 2021

Acknowledgements

I acknowledge and dedicate this thesis to my family that has been of great support through the three years of this study. I also acknowledge the efforts of my mother and late dad, Esther and Francis Kiruki, for laying a foundation for my academic and research journey.

I acknowledge my colleagues in the department of Electrical and Space Systems Engineering that have made my journey at Kyutech worthwhile. Particularly, I acknowledge all the team members of the two satellite projects that I was involved in: Tenkoh and Kitsune. Their dedication and efforts were very helpful to me in gaining hands-on experience on nanosatellite design and construction.

I also acknowledge the Government of Japan and UNOOSA PNST Fellowship Program for granting me a PhD scholarship that has enabled me to pursue this study and research. Without this fellowship, I would not have had such a great opportunity in life.

Finally, but not the least, I acknowledge the immense contribution of my supervisor, Prof. Kenichi Asami in this study and research. He has offered guidance and support to me in the course of my three years here in Kyutech. I also acknowledge the contribution of Prof. Mengu Cho (Kitsune) and Prof. Keichi Okuyama (Tenkoh) for their guidance on nano-satellite projects as well as greater exposure in the space field.

List of Figures

List of Tables

Contents

Abstract

This thesis is focused on spacecraft onboard inference. The advancement in artificial intelligence has opened new possibilities for application in the space field. While some of these applications are terrestrial, some are in on-orbit operations. Hence, there is need for hardware designs and architectures capable of efficient inference onboard the spacecraft. We commence this thesis with literature review focused on the three main themes of this work.

The first theme is on machine and deep learning for space applications. Such applications are in space robotics, satellite missions and navigation. One of the growing research areas for adoption of artificial intelligence in space applications is in navigation, particularly pose estimation in monocular vision-based navigation. This is beneficial in both satellite missions and in outer space explorations. For example, in Mars and asteroid explorations, the rovers benefit from pose estimation in their autonomous navigation and operation. Spacecrafts also utilize pose estimation in on-orbit operations such as servicing, docking, debris removal and other rendezvous operations. The second theme is on investigation of suitable hardware for onboard inference in space applications. From the literature review, FPGAs are found to be suitable candidates due to their parallelism that is amenable for inference. They are also low-cost and low-power and therefore can be utilized in relatively small spacecrafts. This forms the third theme of this work, which is FPGA-based inference accelerators. Xilinx FPGAs are utilized in implementing the onboard inference in two approaches: custom-based and IP-based.

This thesis is organized as follows. Chapter 1 briefly introduces the thesis work. In chapter 2, a detailed literature review is presented. This includes the different areas in space field in which artificial intelligence and autonomous approaches have been utilized. It also highlights current research works in the field and possible applications of AI in the space field and environment. The chapter also presents the different challenges facing these space applications in comparison to terrestrial applications. It also presents the different computing hardware that has been employed in such space applications including in planned and future missions. Chapter 3 presents a brief theory on the three main themes of this research work.

One of the themes is FPGAs, in which the Xilinx family of FPGAs has been utilized in this work. Hence greater attention is dedicated to these devices.

Chapters 4, 5 and 6 present the methodology carried out in this work. Chapter 4 covers the implementation of the CNN part of a CNN-based spacecraft pose estimation. This is the spacecraft keypoints/landmarks localization. This chapter presents the dataset adopted for the network training, architectures of the various convolutional neural networks considered, and the approach taken in the training and evaluation phase. Chapter 5 presents the first approach for FPGA-based inference. In this design, a custom 3-layer multi-layer perceptron (MLP) network is implemented. The implementation was simulated and tested on the Kintex-7 device. In this implementation, the major design considerations were the activation function implementation, data precision and parameterization. Whilst this approach was verified, it had shortcomings in the sense that it is not easily scalable to deeper networks, is time consuming and prone to errors.

Chapter 6 presents the IP-based approach to the inference goal. This makes use of the Xilinx Deep Learning Processor Unit (DPU) IP. This IP is optimized for deep learning and can be configured to use a variety of network models and architectures. It also supports networks that are trained in most of the dominant frameworks such as TensorFlow. This approach requires the need for a hybrid computing architecture. It utilizes Programmable Logic (PL) and Processing Systems (PS). The DPU IP is instantiated in the PL side and is operated by system calls and functions from the PS side. This hybrid architecture inherently requires an underlying operating system, Linux. In this implementation, the Xilinx Petalinux and Yocto are used to achieve the DPU implementation and operation.

The thesis completes with a presentation of the results in chapter 7 achieved in the two approaches. It also discusses the merits and demerits of the approaches. Chapter 8 offers a brief discussion and conclusion of this thesis, including possible future research direction. It is also acknowledged that this is a fast-changing environment and other hardware possibilities will inadvertently arise in future. More so, space data will increasingly become available and this will open opportunities for application of deep learning solutions in a wider spectra of space fields.

1 Introduction

Recent activities in the space field have revolutionized the old space field. This has led to a new space age that has been labeled as Space 2.0. The old space was dominated by government national agencies that were the main actors in space exploration. This was mainly due to the extremely high financial and technical capacities required in most of the space activities. However, the new space age is ushering in players from the private and research sector. This has been facilitated by improvement and subsequent lowering of costs associated with many space sectors such as satellite construction and launch costs. Technology development has led to this lowering of costs as missions previously requiring big spacecraft can now be achieved by relatively smaller spacecraft. This has led to development of new business models cutting across the various sectors in the space industry. For example, hundreds of satellites have already been launched into the Low Earth Orbit for provision of broad-band internet connectivity. These are constructed and deployed by private enterprises.

This opening up of space has gone beyond the Earth's orbit. The Moon and Mars are the next frontiers in the new space age for exploration. All these developments require and encourage diverse missions and new technology to carry out missions that are continuously growing in complexity. This need is filled by technology that has also been growing terrestrially. While the space field is witnessing a Space 2.0 fueled by technology and private enterprises, terrestrially, artificial intelligence has opened new frontiers. AI has facilitated various aspects of the day-to-day activities. This includes being a dominant technology in research fields such as autonomous driving systems. Other notable terrestrial applications have been in natural language processing, speech recognition and medical images analyses [1].

Naturally, the two worlds of Space2.0 and AI have converged by creating new applications as well as needs. In [2], Kothari et al (2020) explore latest areas that have benefited from machine learning and categorize them into two domains: analysis of payload data and general spacecraft operation. In the former, the areas include weather and atmospheric monitoring, vegetation and ground cover classification, and object detection and tracking. In the domain of spacecraft operation,

the areas include communication, automated control, and navigation. One of the areas to experience this convergence is in earth observation and remote sensing. AI has increased utility in the field by enhancing processing of huge remote sensing data and deriving inferences from this data. While most of these applications have been confined to terrestrial systems, there is more focus on deploying such and similar applications onboard spacecraft themselves. This is partly driven by the demand imposed on Space 2.0 spacecraft in terms of complex missions in small spacecraft that need to collect and downlink huge amounts of data, especially in earth observation missions. Consequently, AI is increasingly being used or proposed for application in areas such as navigation in Martian terrain, onboard data processing, pose estimation in rendezvous operations and in space debris mitigation and removal.

1.1 Problem Statement

Though AI is increasingly being adopted in various space areas, there are differences in terrestrial and space applications which pose a challenge to adaptability of terrestrial approach to space implementations. For example, terrestrial application of AI is facilitated by shared resources such as cloud services, powerful computing hardware and machine learning framework APIs. However, these enablers are limited in the space environment. Specifically, utilization of deep learning solutions is limited in space operations due to two key factors. Deep learning requires huge data sets for training. Whilst space data is increasingly available, many fields in space still lack sufficient data for training deep networks in these niche areas as compared to the vast amount of data available in terrestrial applications. Nevertheless, there has been innovative ways to address this lack of sufficient space data for training. Transfer learning is one of the most promising approach to address this shortfall [3]. It utilizes general data (terrestrial) to train most of the deep layers in the neural networks.

Another challenge of deep learning in space applications is related to hardware limitation. Terrestrially, variety of computing hardware is extensive and powerful. General purpose computers, GPUs and cloud services are available for terrestrial applications. More so, there is an emergence of deep learning custom hardware col-

lectively referred to as Neural Processing Units (NPUs). In [4], Microsoft presents a configurable cloud-scale DNN processor whilst in [5], Google presents an in-datacenter performance analysis of their custom Tensor Processing Unit (TPU). These kinds of hardware are targeted at terrestrial applications that inherently are power-hungry and therefore not suitable in power-constrained spacecrafts. This limits their adoption to space applications especially in micro satellites that are limited in terms of power and size budgets.

Therefore, as briefly introduced in the preceding paragraphs, the adoption of AI technology onboard spacecrafts is greatly limited by space hardware. Hence, to take advantage of the progress in AI, space applications will need to have efficient and suitable hardware for onboard inferencing. Two of the main platforms suitable for machine learning computations are Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). However, GPUs are power hungry compared to FPGAs for similar operation and accuracy. Therefore, FPGAs are more suitable in space applications. More so, FPGAs are already being used in space applications in mission-specific and satellite subsystem operations. They are also re-configurable, and therefore can be adopted for different tasks on the fly.

One of the areas that deep learning technology has been proposed as a suitable addition is in navigation both in outer space exploration and rendezvous operations. To facilitate this, pose estimation is a fundamental requirement of such systems. Deep learning has proved to be effective in terrestrial pose estimation applications. This has led to its proposed utilization in space applications. A challenge, especially for small spacecraft, is the availability of suitable hardware for onboard inference of the deep learning networks.

This thesis is geared towards investigating the application of FPGAs as a suitable platform for onboard inference. It investigates the design, development and deployment of deep learning networks on an FPGA in the area of pose estimation. The Xilinx FPGAs are utilized in this study. Two approaches are presented in the methodology: custom-based and IP-based inference.

1.2 Thesis Outline

The major focus of this thesis is the utilization of FPGA as a platform suitable for on-board inferencing. Whilst there are several vendors of FPGA devices, Xilinx is the dominant player in this market and its tools and devices have been used in this work. The Kintex-7 and UltraScale MPSoC devices have been utilized for the two implementation approaches. In the first approach, the custom implementation was based on the basic building blocks of FPGA fabric. Hence this implementation is extremely custom and requires implementing the network ground-up. Kintex-7 device was used as the target device. Though this method was amenable to greater optimization since the user has greater freedom of choice on many network parameters, it was found to be time-consuming and prone to errors. This is because FPGAs have stringent timing requirements and as the network size grows, these needs become difficult to meet. In the second approach, a vendor IP was used as the building block for the inference solution. The Xilinx Deep Learning Processor Unit (DPU) IP was used to implement a Convolutional Neural Network (CNN). This IP is optimized for various deep learning architectures and is highly parameterized as well. Nevertheless, it is still in early development phase and its capabilities are in active improvement phase.

This thesis is organized as follows. Chapter 1 is a brief introduction to the thesis body of work. In the second chapter, a detailed literature review is presented. This includes the different areas in space field in which artificial intelligence and autonomous approaches have been utilized. It also highlights current research works in the field and possible applications of AI in the space field and environment. The chapter also presents the different challenges facing these space applications in comparison to terrestrial applications. It also presents the different computing hardware that has been employed in such space applications including in future and planned missions.

Chapter 3 presents a brief theory on the three main themes of this research work. One of the themes is FPGAs. This includes their architecture and use cases. The Xilinx family of FPGAs has been utilized in this work. Hence greater attention is dedicated to these devices. The two devices that have been used in this work are:

Kintex-7 and Ultrascale MPSOC. Their architecture and features are presented in this work. In addition, the development cycle of FPGA-based solutions is presented in this chapter. This covers the software and hardware tools required in such implementations. The focus is on Xilinx-based tools, including Vivado, Petalinux, SDK and Vitis. The other theme is CNNs. Some of its concepts including convolution layers, kernels and pooling are introduced. The third theme whose brief theory is presented is pose estimation.

This thesis methodology is covered in chapters 4, 5 and 6. Chapter 4 covers the keypoints localization implementation, which is the CNN part of a CNN-based Pose Estimation for a spacecraft. It presents the dataset adopted for the network training, architectures of the various convolutional neural networks considered and the approach taken in the training and evaluation phase. The training data is based on the European Space Agency (ESA) Pose Estimation Challenge. Chapter 5 presents the first approach for FPGA-based inference. In this design, a custom 3-layer multi-layer perceptron (MLP) network is implemented. This utilizes basic building blocks of the FPGA logic such as Look-Up Tables (LUTs), Block RAMS, Digital Signal Processing engines (DSPs), Flip Flops (FFs) and other logic elements. The implementation was simulated and tested on the Kintex-7 device. The MNIST dataset was used to train and test the network. In this implementation, the major design considerations were the activation function implementation, data precision and parameterization. Whilst this approach was verified, it had shortcomings in the sense that it was not easily scalable to deeper networks. Since it is a customized approach, it is time consuming as well as prone to errors.

Chapter 6 presents the IP-based approach to the inference goal. This makes use of the Xilinx DPU IP. This IP is optimized for deep learning and can be configured to use a variety of network models and architectures. It also supports networks that are trained in most of the dominant frameworks such as Caffe and TensorFlow. This approach requires the need for a hybrid computing architecture. It utilizes Programmable Logic (PL) and Processing Systems (PS). The DPU IP is instantiated in the PL side and is operated by system calls and functions from the PS side. This hybrid architecture inherently requires an underlying operating system, Linux. In this implementation, the Xilinx Petalinux and Yocto are used

to achieve the DPU implementation and operation.

The thesis completes with a presentation of the results in chapter 7 achieved in the two approaches. Chapter 8 discusses the merits and demerits of the approaches. In this discussion, possible future research direction is presented. It is also acknowledged that this is a fast-changing environment and other hardware possibilities will inadvertently arise in future. More so, space data will increasingly become available and this will open opportunities for application of deep learning solutions in a wider spectra of space fields.

1.3 Summary on Contributions of this Work

This work has presented a survey on the hardware suitable for inference in onboard space applications. It focuses on both hardware with flight heritage as well as that in planned and future missions. The thesis also surveys some of the space applications in which artificial intelligence and autonomous operation can be utilized.

One contribution of this work is a modularized and flexible neural network inference implementation on an FPGA. This approach implements the inference in a modular approach to allow easy integration of different design architectures. It is also flexible in that different modules can be swapped easily to improve the efficiency and performance of the network. For example, various hardware designs of the activation function can be implemented. Due to the modularity and flexibility, they can be easily switched depending on the performcnce requirements as will be depicted in chapter 5.

The other contribution is on the IP-based inference. This is the major focus for this work. FPGA was identified as a suitable candidate for spacecraft onboard inferencing. In 2019, Xilinx introduced the Deep Learning Processor Unit IP that enhanced the utilization of its devices for inference. The multiprocessor system on chip (MPSoC) devices are targeted for this IP implementation. In this work, a CNN utilized in pose estimation is implemented on the Zynq MPSoC that contains both FPGA fabric and processor cores. The thesis presents the design process, commencing with network training, post-processing for inference and the hardware implementation phase.

2 Literature Review

This section will present a review of the state-of-art in areas surrounding and influencing adoption of AI to space applications. The section focuses on three parts. First part is an overview of the space fields in which machine and deep learning algorithms can or have been utilized. It presents some of the recent developments in the sub-field of machine learning for onboard space applications and not the entire AI field. Hence it's limited to in-flight usage as opposed to ground-based operations which have been covered widely in studies such as [6] where deep learning has been utilized in remote sensing for land cover and crop type classification whilst David et al in [7] reviewed the use of machine learning in geosciences and remote sensing. Due to the diverse and enormous corpus of machine learning materials and applications, this review section is not an exhaustive in coverage of all such applications but gives an overview and general state of the art.

The second part focuses on the hardware available for on-board inferencing. This concept of orbital edge computing for space applications has been introduced in [8] which argues for more data processing on-board a satellite. Edge computing refers to placing processing hardware near data sources, in this case the satellite. It will also explore various techniques and solutions available for efficient on-board machine learning inference. The last part will focus on one of the areas that deep learning is very amenable for adoption in the space field, pose estimation.

2.1 Application of AI in Space Autonomous Operations

The number of small and micro-satellites launched are increasing. There is also much data that is being accumulated in these missions. More so, these satellites are mostly in a standardized form such as CubeSats. Others such as those for LEO internet services proposed by SpaceX and other companies are similar in operations and structure [9] [10]. Therefore, the efficiency of these satellites can be increased by making use of the data accumulated by similar satellites in similar orbital conditions. Newer satellites replacing de-commissioned ones in the constellations can use their accumulated data for tasks such as on-board health status monitoring and fault detection. This relieves ground stations the need to analyse huge amounts

of satellite housekeeping data, hence creating more communication bandwidth for the satellites. In some applications, satellite and space missions require a level of autonomous operation. These include motion-planning for planetary rovers and robotic SmallSats such as that presented in [11]. There are numerous proposals for the application of AI techniques in various space fields, some of which have already flown in space. This section presents some of these applications.

Girimonte and Izzo in 2007 presented one of the earliest reviews on the application of artificial intelligence in the field of space engineering and technology [12]. They focused on three areas of interest: distributed artificial intelligence, enhanced situation self-awareness, and decision support for spacecraft system design. In distributed AI, the paper proposed use of swarm intelligence for autonomous operation of satellite constellations such as motion planning and coordination. To enhance self-awareness, the paper posits that a spacecraft should be able to perform autonomous actions and determine its own health status. This can be achieved by data-driven approaches such as ML. These algorithms can make use of in-flight time-series data that can lead to autonomous identification of system faults and errors.

Over the last decade since the publication of their work, there has been an explosion in the field of general AI. Other related works since then have narrowed their focus to application of AI for specific space areas. Jeremy in [13] reviews use of AI in spacecraft control systems. The work presented various AI space applications of the 1990s and early 2000s. These included the Deep Space 1 (DS1) craft that was controlled by an autonomous agent known as Remote Agent Experiment (RAX). Another significant application was the Autonomous Sciencecraft Experiment (ASE) onboard the Earth Observing-1 [14]. This equipped the spacecraft with onboard analysis to detect phenomena of interest such as floods, volcanos etc and transmit the most valuable information to earth. NASA's Astro and NextSat docking and rendezvous were also highlighted.

This section identifies and groups the fields in which AI-related technology has been utilized in space applications into three key areas namely: space robotics, satellite missions and spacecraft guidance and control.

2.1.1 Space Robotics

Outer space and planetary explorations have relied heavily on robotic operations. These include Mars and asteroids missions. The Mars Rovers (Sojouner, Spirit, Opportunity and Curiosity) have been instrumental in mapping and navigating the Martian terrain [15]. AI has been a critical component of these rovers by enabling them to operate autonomously. This autonomy includes navigation, target identification, automatic detection of interesting science events, amongst other manoeuvres [16] [17]. This autonomy capability was greatly enhanced by incorporation of Autonomous Exploration for Gathering Increased Science (AEGIS) software aboard Curiosity rover in May 2016. This software enables intelligent image analysis to find targets in images taken by the rover's stereo navigation cameras (NavCam). This allows the rover to autonomously scan its environment, identifying and ranking the best bedrock to study with its ChemCam which is an instrument for determining the chemical composition of rocks. This autonomous targeting facilitates the rover to navigate into new unknown areas and acquire ChemCam observations of targets not yet seen by scientists on Earth [18].

Apart from planetary explorations, machine learning and robotics has been utilized in On-Orbit Operations (O3). Such operations include on-orbit servicing of spacecraft (inspection, repair), on-orbit assembly of modular systems such as ISS and large aperture telescopes and active debris removal. In 2007, NASA demonstrated on-orbit servicing using two satellites: ASTRO (servicing satellite) and NextSat (serviceable satellite) [19]. Angadh et al in [11] investigate the use of robotics and AI for O3 operations using small satellites. They highlight some of the technical challenges that exist in robotic O3 such as: pose estimation of chaser and target; autonomous rendezvous and docking manoeuvres. The work also gives a detailed review on past and planned space robotics targeted at O3 operations.

2.1.2 Satellite Missions

Some form of artificial intelligence has been utilized in specific missions aboard satellites and spacecrafts. These applications could be in onboard data processing or autonomous satellite operations. The Deep Space 1 (DS1) launched in 1998 in-

corporated an autonomous remote agent to plan and execute spacecraft activities. The agent could autonomously detect and diagnose faults. It could then resolve or work around the detected problems without requiring help from the ground [20]. This remote agent included an on-board mission manager that carried the mission plan, expressed as high-level goals. These were then used by a planning and scheduling engine to generate a set of time-based or event-based activities that were delivered to the executive. The latter would then expand them into a sequence of commands and issue them directly to the appropriate spacecraft systems. In 1999, the remote agent controlled the spacecraft for approximately two days [21]. DS1 also incorporated autonomy in monitoring the overall spacecraft health. This onboard system would then transmit one of four tones needed for beacon monitoring. These tones indicated the level of urgency of the spacecraft's need for the Deep Space Network (DSN) coverage. Hence the large and expensive DSN stations were only used for critical DS1 operation based on the tone transmitted by the onboard AI monitoring system [22].

The Earth Observer 1 (EO-1) satellite launched in 2000 (was decommissioned in 2017) had a technology demonstration mission that included the Autonomous Sciencecraft Experiment (ASE) software [14]. This was an onboard intelligent system that enabled the satellite to make autonomous decisions based on data observations or other events. Hence it could schedule imaging choices on detecting phenomena such as floods, volcanoes etc [23]. ASE was designed in a way that it could be upgraded from the ground to improve its capability. Hence, over the years the system was continuously improved. Some of the onboard science-based data analysis capabilities included thermal anomaly detection to identify volcanic eruptions, support vector machines (SVMs) for cryosphere features classification and detection of sulphur deposits on glaciers etc. Kiri et al in [24] incorporated additional ML onboard the EO-1 spacecraft for data analysis to optimize use of limited downlink. They implemented cloud detection with both Random Decision Forest (RDF) and Bayesian Thresholding. This enabled EO-1 to filter and discard the Hyperion images that contained obscuring clouds. This ensured best use of the limited downlink.

With the growth and maturity of CubeSat technology, NASA launched the In-

telligent Payload Experiment (IPEX) CubeSat in December 2013. This was a technology demonstration platform for validating onboard autonomous operations of the Intelligent Payload Module of the Hyperspectral Infra-red Imager (HyspIRI) mission concept [25]. IPEX incorporated various image processing techniques such as SVMs and RDF classification algorithms. The latter was trained prior to launch using just four hand-labeled images from a high-altitude balloon flight. It achieved accuracies of over 95 % for a four-pixel categories classification: (i) clear surface (land or ocean); (ii) planetary limb; (iii) clouds; and (iv) outer space. IPEX achieved its full mission criteria which included demonstrations of autonomous onboard product generation and autonomous payload operations [26].

With the recent advancements in machine learning, custom hardware incorporating ML features are being developed. Such a recent hardware for space applications is the Deep Learning Attitude Sensor (DLAS) [27]. This is a low-cost star tracker and Earth sensor developed by Tokyo Tech and installed on JAXA's RAPIS-1 satellite that was launched in January 2019. Using two cameras incorporated in the star tracker baffle, real-time image recognition using deep learning is achieved. This recognition of vegetation and land use is under nine categories including green terrain, oceans, deserts, clouds, and outer space. The algorithm is based on a simple multi-layer perceptron (MLP) that used images captured by ISS as the training dataset. Relearning was conducted using images captured by DLAS after launch. This deep learning-based image recognition is also used for estimating 3-axis attitude using land features obscured by clouds and comparing it with map data pre-recorded in the onboard computer.

The above highlighted cases are some of the missions incorporating machine learning techniques that have been flown in orbit. There are many other novel AI-based missions that have been proposed but this review was confined to already flown missions. The next section presents use of ML in spacecraft navigation. This can also be considered as a satellite mission in some cases. However, this study tries to make a distinction between navigation missions and other onboard missions.

2.1.3 Navigation and Pose Estimation

Determination of the orientation and position of a spacecraft is an important aspect in various space operations. One of the areas that this is critical is in debris removal applications. There has been increased space activity with more participants including universities, private companies, and research organizations accessing space; especially with the rise in popularity of CubeSats and nano-satellite technologies [28]. Satellite internet services have also gained momentum with thousands of microsatellites set to be launched in low earth orbits for provision of global broadband. The increased space activities have led to fears of an increase in space debris [29]. To address such concerns, studies for space debris removal have been conducted [30] [31] [32], in which effective pose estimation is required. On-orbit servicing and assembly of spacecrafts is another field where pose estimation is fundamental [33]. In [19], NASA demonstrated on-orbit servicing using two satellites: ASTRO (autonomous servicing satellite) and NextSat (serviceable satellite). Nanjangud et al in [11] present the use of robotics and AI for on-orbit operations (O3) using small satellites. They highlight some technical challenges in robotic O3 such as: pose estimation of chaser and target; autonomous rendezvous and docking manoeuvres.

Another space area where pose determination and navigation capabilities are required is in outer space and planetary explorations. The Mars Rovers (Sojouner, Spirit, Opportunity and Curiosity) have been greatly utilized in Mars exploration missions. These rovers have incorporated autonomous operations including navigation, target identification and automatic detection of interesting science events [16] [17]. For such autonomy capability, the rovers need to accurately determine their position and orientation within the Martian terrain. This aids in their navigation as well as identifying and approaching their targets for scientific experiments. These manoeuvres are greatly aided by reliable pose estimation solutions. Docking of spacecrafts and other rendezvous operations such as formation-flying are also reliant on effective pose determination. This fundamental need for reliable pose estimation is underlined by the organization of Pose Estimation Challenge in 2019 by European Space Agency (ESA) and Space Rendezvous Laboratory (SLAB) [34].

Among various pose estimation methods, CNN-based algorithms have recently proved to have better performance and are attracting more research focus. In this paper we present how such CNN-based approaches can be realized on resource constrained (power and size) spacecrafts such as small satellites and space robotics. The paper is focused on onboard implementation of the CNN-part of such algorithms. A hybrid FPGA-CPU device, Xilinx's UltraScale+ MPSoC, has been chosen and proposed as the target device. In CNN-based algorithms, the CNN inference requires greater computation power and resources that typical processors cannot meet for real time operation. On the other hand, the Programmable Logic (PL) of FPGA is very suitable as an inference engine. The focus of this paper is only on the inference part. In future work, the final pose estimation part of the algorithm will be implemented on the CPU part of the hybrid device. To the best of the authors' knowledge this is the first attempt of such onboard inference focusing on the CNN-part of the spacecraft pose estimation challenge.

2.2 Hardware Inference for Space Applications

While various space applications, including pose estimation, can benefit from deep learning, they face challenge in suitable hardware for inference. Small satellites have limited power, mass, and cost budgets, which poses computing power challenges compared to terrestrial applications. The space environment also offers challenging conditions for the operation of electronics. Lu et al. in [35] review the effects of space environment in different orbits. They observe that a spacecraft is mainly affected by the following space components: neutral atmosphere, plasma, radiation, macroscopic particles, geomagnetic, and temperature fields. LEO is predominantly influenced by radiation and macroscopic particles, whilst MEO and GEO are greatly exposed to solar activities, plasma environment, and radiation. Approximately, radiation anomalies and temperature-induced anomalies account for 40% and 11% of spacecraft failures in the space environment, respectively [35].The main radiation sources in space include trapped radiation, galactic cosmic rays, and solar energy particles.

Damage to electronic devices on board spacecraft due to radiation is mainly categorized into ionizing and non-ionizing (displacement) damage. Displacement

damage is the cumulative long term non-ionizing damage due to protons, electrons, and neutrons. Ionization damage is the creation of electron-hole pairs and it can be further categorized into total ionizing dose (TID) and single event effects (SEE). The former is the cumulative long term ionizing damage due to protons and electrons, whilst the latter is caused by a single charged particle such as heavy ions and protons [36, 37]. The Tenkoh satellite that was developed at Kyushu Institute of Technology and launched in October 2018 was analyzed to have suffered SEEs after passing over the South Atlantic Anomaly [38] .

Due to these harsh conditions in space, radiation-tolerant, radiation-hardened, and space-grade computing hardware is desirable for space operations. This has limited the options available for hardware suitable for inference in space. Nevertheless, there is hardware with flight heritage that is very amenable to on-board inference. This section explores such computing hardware that is based on flight heritage or planned missions, support for CNN inference, and other factors, such as low power consumption. These are summarized in Table 1.

This survey reveals that hybrid computing devices comprising FPGAs and Systems-on-Chips (SoCs) are preferred for implementing machine and deep learning algorithms onboard spacecraft. However, most of these are not space-grade devices. George et al. in [39] identified hybrid and reconfigurable computing as driving the revolutionary capabilities of small satellites such as cubesats. Lentaris et al. in [40] conducted a review of high-performance embedded computing for vision-based navigation in space based on four categories: FPGA, CPU, GPU, and DSP. They found that FPGAs achieved the highest performance per watt of all platforms by at least one order of magnitude. GPUs are power hungry when compared to FPGAs and CPUs for comparable operation and accuracy [40]. CPUs performance fade in comparison to FPGAs in DL inference. Consequently, FPGAs are more suitable for space inference-based applications. Additionally, they are already being used in mission-specific and satellite subsystem operations. They are also reconfigurable and, therefore, can be adopted for different tasks on the fly.

Table 1: Summary of Hardware Suitable for Inference in Space.

Device	**Vendor**	**Flight Heritage**	**Inference Support**
Zynq-7000 SoC	Xilinx	CSPv1 (ISS, 2017), Cubesat Space Processor (Space Micro), NanoMind Z7000 (GOMspace)	Xilinx Edge AI Platform, Xilinx Machine Learning Suite, Xilinx AI Inference Acceleration
Movidius™ Myriad 2 Vision Processing Unit	Intel	PhiSat-1 (ESA, Sept 2020)	Neural Compute Engine, OpenVINO™ toolkit, Myriad Dev Kit (MDK)
Kintex® UltraScale™ XQRKU060	Xilinx	Radiation-Tolerant(No Flight Heritage)	Xilinx Machine Learning & AI Inference Acceleration
UltraScale+ Zynq MPSoC	Xilinx	Defense-Grade (No Flight Heritage yet)	Xilinx Deep Learning Processing Unit (DPU) IP, Vitis-AI Platform
DAHLIA SoC (arm Cortex-R52 & FPGA)	European Consortium	Planned (2020)	Arm Machine Learning Processor, ArmNN SDK, Arm Compute Library(ACL)
Chiplet SoC(arm® A53 Processors)	NASA	Planned (2021)	Arm Machine Learning Processor,ArmNN SDK,(ACL)
RTG4 FPGA	Microchip (Microsemi)	Jena-Optronik LIDAR Sensors (ISS Supply Vehicles Docking)	N/A
SmartFusion FPGA	Microchip (Microsemi)	UNSW-EC0 CubeSat (Australian, 2017)	N/A
Sitara TM Processors	Texas Instruments	Gumstix COMs aboard DM7 Experiment on ISS	Texas Instruments Deep Learning (TIDL), ArmNN software frameworks
LS1046 Microprocessor	NXP	Teledyne e2v Radiation Tolerant Space Microprocessors(Planned 2021)	NXP® eIQ™ ML Software Development Environment, ArmNN

In May 2020, Xilinx unveiled the Xilinx® Radiation Tolerant (RT) Kintex® UltraScale™ XQRKU060 FPGA. The device is optimized for various computational-intensive space applications. It is the first 20 nm space-grade FPGA optimized for machine learning inference coupled with unlimited on-orbit reconfiguration for real time on-board-processing. Some of its key applications include on-board AI for autonomous space exploration, real-time streaming of earth observation, remote sensing video, and for flexible, digital beam-forming telecommunication satellites [41]. In September 2020, ESA launched the Phi-sat-1 satellite that incorporated AI for Earth observation. Intel's Myriad 2 Vision Processing Unit was used as the onboard inference platform, although it is not space-grade. This hardware was incorporated to utilize deep CNN in automatic cloud cover identification. Synthetic data from existing missions were used as the training data. It is the first European satellite to demonstrate how onboard artificial intelligence can improve the efficiency of sending EO data back to Earth. Initial data downlinked from the satellite showed successful assortment of the hyperspectral imagery into cloudy and non-cloudy pixels [42]. This preceding hardware survey informed the decision to pick the Xilinx Zynq UltraScale+ MPSoC for onboard CNN inference. This is because its programmable logic can be comparable to the XQRKU060 FPGA logic, as presented later in Table.

2.3 FPGA-Based Inference Accelerators

In terrestrial applications, FPGAs have been investigated and utilized as inference accelerators in deep and machine learning applications. The inherent parallelism of FPGAs has been instrumental in achieving real-time inference for such applications. The evaluation boards used in FPGA-based inference have hybrid CPU + FPGA heterogeneous architecture. The Programmable Logic (PL) is the FPGA chip and it contains the computing complex, processing elements, on-chip buffers, controller, and the DMAs. The Processing System (PS) consists of the CPU and external memory. Most of the FPGA-based accelerators have been specific to different CNN architectures and models. Figure 1 shows a typical accelerator that is based on FPGA. The CNN inference is performed in the PL side, whilst a CPU is required for pre-and post-processing and scheduling tasks. An external memory is

required to store the CNN model parameters, data, and instructions. A full CNN model comprises of both convolutional (CONV) and fully-connected (FC) layers. The former are computational-intensive, whilst FC layers are memory-centric, as they typically contain millions of weights. Basic architectures of FPGA-based accelerators can be grouped in three categories: (i) single processing engine, usually in the form of a systolic array that processes each layer sequentially; (ii) streaming architecture that consists of one Processing Element (PE) per network layer; and, (iii) vector processor with instructions that are specific to accelerating the primitive operations of convolutions.

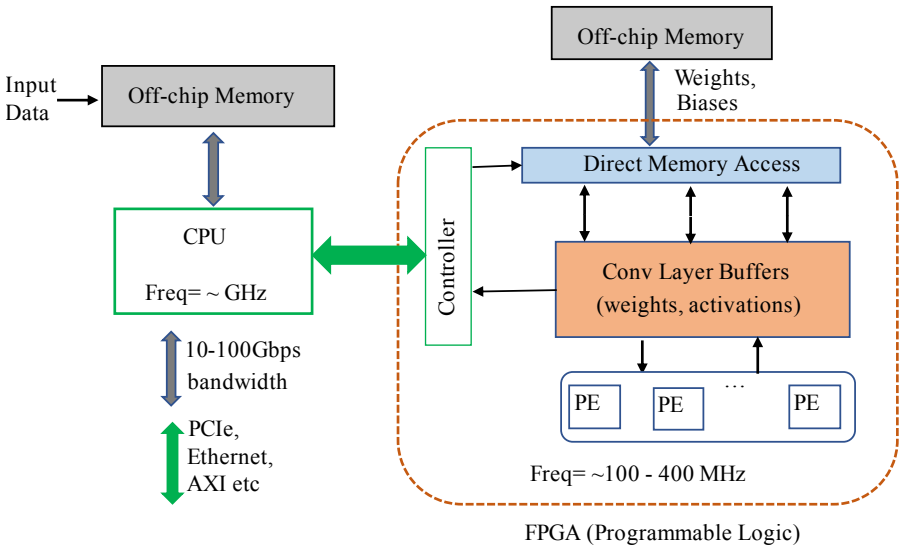


Figure 1: Typical FPGA-based Inference Architecture.

Guo et al., in [43], conducted a survey on FPGA-based NN inference accelerators. They note that the main considerations for such accelerators is high speed (high throughput and low latency) and high energy efficiency. One of the key steps in adopting models for FPGA inferencing is reducing the network size i.e., model compression. This can be achieved by approaches such as data quantization and weight reduction, via methods such as pruning. The survey also explores the various hardware design methodologies that have been adopted for efficient architecture, including computation unit designs, loop unrolling strategies and the overall system design taking into account CPU, FPGA and memory configurations.

It also investigates the different automation approaches for mapping networks to hardware i.e., hardware and software design automation.

Dinelli et al. presented one example of hardware design methodology in [44], where they implemented a MEM-OPT system to address the on-chip memory bottleneck of FPGA-based hardware accelerators. The MEM-OPT system is composed of three design aspects: a scheduling algorithm, a Secondary Cache System (SCS) for data re-use and support for different configurations. The scheduler determines the number of elements read out of the Input Cache (IC) for efficient on-chip memory usage. The SCS memory enables IC data re-use, hence reducing the amount of data read out the IC, because successive convolution operations share part of the input data previously read. The MEM-OPT does not require output buffer, because each processing element computes only one output value at a time. This system showed considerable reduction of on-chip memory (BRAMs) usage when compared to other scheduling algorithms.

Wei et al., in [45], implemented CNN on FPGA while using systolic array architecture for high throughput. This approach ensures low global data transfer since the processing elements (PEs) do not need to access the on-chip memory. Instead, connections are only required between different computation units for data transfer. This systolic architecture enables achievement of high frequency even in massive parallelization with hundreds of PEs. They also implemented an automated flow to map CNN architectures from high-level C code to FPGA, with no hardware-related, low-level considerations necessary for end-users. Lian et al. in [46] implemented an FPGA-based CNN model by adopting an optimized block floating-point (BFP) arithmetic. The BFP is composed of a mantissa part, whose bit length is defined as 8 for typical CNN models, and an exponent part. Quantization involves both FP2BFP and BFP2FP conversions.

Earlier quantization strategies had gone to as low as binary representation. Courbariaux et al., in [47], introduced a method of training such Binarized Neural Networks (BNNs). This enabled the use of binary weights and activations for computing the parameters gradients. This enables replacement of arithmetic operations with bit-wise operations, even during train-time, hence reducing training time, memory

consumption, and increasing power-efficiency. During training, the weights and activations are constrained to either $+1$ or -1. Rastegari et al., in [48], implemented two efficient binary variations of convolutional neural networks. The first one, Binary Weight Networks, had all the weight values approximated with binary values whilst the second one, XNOR Networks, had both weight and input with binary values. The former led to 32x smaller networks when compared to equivalent networks with single-precision weight values and resulted in 2x speed up. The XNOR-Nets resulted in 58x speed up, whilst offering accurate approximation of CNNs. Umuroglu, in [49], presented a framework (named FINN) for building scalable and fast inference accelerators on FPGAs. They provide an end-to-end mapping of BNNs onto FPGAs. The FINN implementation uses separate compute engines that were dedicated to each layer and which communicate via on-chip data streams. Each engine starts to compute as soon as the previous engine starts to produce output. This framework was further improved into FINN-R that automated the creation of fully customized inference engines for quantized neural networks [50].

One of the seminal works on FPGA-based inference accelerators was presented by Qiu et al. in [51]. They implemented a CNN accelerator for the Image-Net large-scale image classification on the Xilinx Zynq ZC706 board.A major contribution was an automatic flow for 8/4 bit dynamic-precision data quantization. They used 8-bit and 16-bit fixed-point numbers for the onboard inference and achieved comparable results to floating-point implementations. This set the foundation for their next contribution in [52], where they present a hardware/software co-design flow for FPGA-based CNN implementation. The original network is compressed to fixed-point representation by optimizing the choice of the radix point positions for the network parameters in every layer. They also implement a parameterized and run-time configurable hardware architecture that supports various networks and that can be adopted on different hardware platforms. They also propose a compiler to map a CNN model to the hardware platform. This work led to the commercial FPGA accelerator engine, DeePhi [53], which was later acquired by Xilinx and improved into the Deep Neural Network Development Kit (DNNDK) package [54].

Previous accelerator engines had to be custom-designed for various networks and model applications. This had made the adoption of FPGAs as inference engine in real-world applications limited. However, the DNNDK package led to Xilinx introducing an Intellectual Property (IP) core, the Xilinx Deep Learning Processing Unit (DPU) [55]. This enhances the implementation of inference engines that are easily adaptable for different CNN architectures and models. Zhu et al., in [56], explored the DPU architecture and design flow by implementing an efficient task assignment framework to maximize performance on DPU-based CNN acceleration. They explored the optimization of task scheduling between the heterogeneous ARM CPU and multiple DPUs. The optimization strategy aims to utilize the DPU in otherwise unused interval time between multiple inference problems running on multiple threads.The optimization strategy also aimed to ensure that the inference tasks of different networks are controllable. The use of DPU has greatly been enhanced by Xilinx's machine learning ecosystem that includes the Vitis-AI and model zoo.

3 Background Theory: FPGAs, CNNs and Pose Estimation

This thesis work is focused mainly on three themes. These are FPGAs, Deep Learning and Pose Estimation. This section covers brief theory and introduction to these three main areas.

3.1 FPGAs

3.1.1 Xilinx FPGAs

Xilinx has a wide variety of FPGA and SoC devices. These FPGA are grouped into families depending on their nanometer scales as shown in Table 2.

Table 2: Xilinx FPGA Families

45nm	**28nm**	**20nm**	**16nm**
Spartan	Virtex-7	Virtex-UltraScale	Virtex-UltraScale+
	Kintex-7	Kintex-UltraScale	Kintex-UltraScale+
	Artix-7		
	Spartan-7		

In this work, two Xilinx devices were utilized. The first is a Xilinx 7 series FPGA and the second device is a multi-processing system-on-chip (MPSoC).The former is contains only the programmable logic whilst the latter has both the programmable logic and the processing system. These two will be presented in the following sections.

3.1.1.1 Kintex-7 & KC705 Evaluation Board

This evaluation board contains the Kintex-7 XC7K325T-2FFG900C FPGA. Unlike the MPSoC device, this has no in-built hard processor. The main features of this board are briefly presented below:

- Memory
  - 1 GB DDR3 memory SODIMM
  - Flash memory: 128MB Quad SPI & 128MB Linear BPI flash memories
- I2C Bus
  - 1KB I2C EEPROM
  - HDMI Codec
  - FMC HPC & LPC connectors
  - User I2C Programmable LVDS Oscillator
- PCI Express endpoint connectivity: Gen1 & Gen2 8-lanes
- 10/100/1000 tri-speed Ethernet PHY
- XADC header

The KC705 Evaluation Board contains the C7K325T-2FFG900C FPGA which is a Kintex-7 and has the following key features as shown is Table 3.

Table 3: Features of the FPGA on-board KC705 Board

Feature	**Description**	**Feature**	**Description**
Logic Cells	326,080	Block RAM 18Kb	890
CLB Slices	50,950	Block RAM 18Kb	445
CLB Distributed RAM Max (Kb)	4,000	Block RAM Max (Kb)	16,020
DSP Slices	840	CMTs	10
GTXs	16	Max User I/O	500

The KC705 evaluation board, block diagram and the Kintex-7 FPGA architecture are shown in Fig. 2.

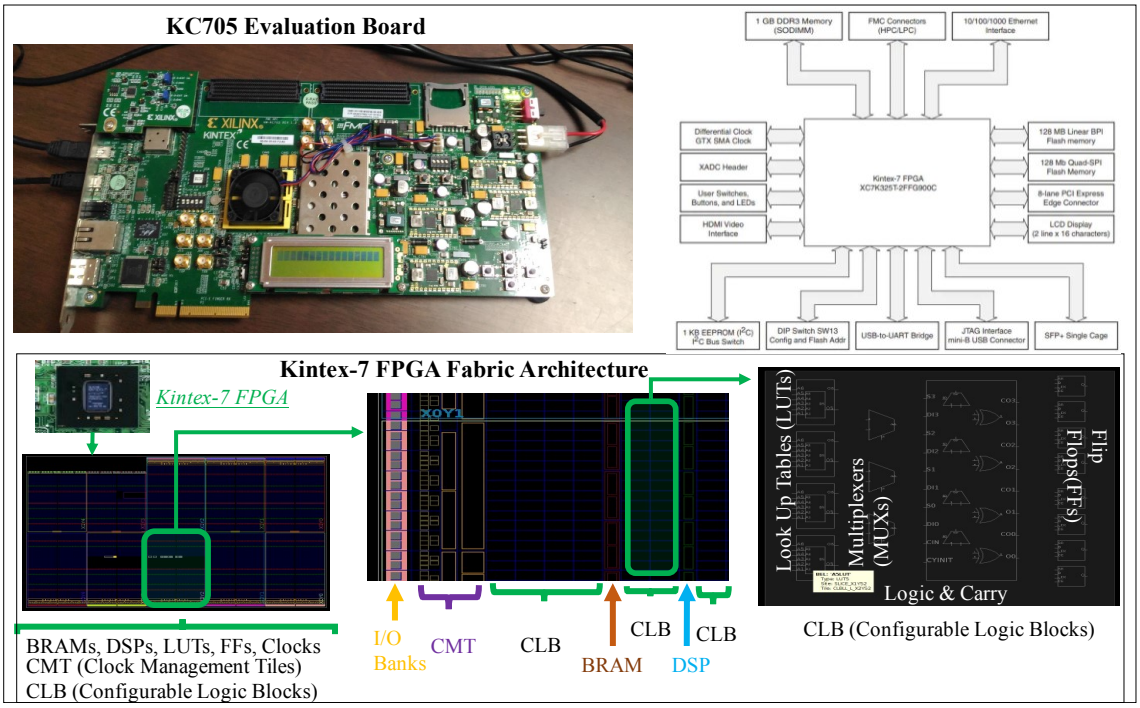


Figure 2: KC705 Evaluation Board

3.1.1.2 Zynq UltraScale+ MPSoC

This device combines FPGA fabric with a processing core. It incorporates both programmable logic (PL) and processing system (PS) on a single chip. Need for Programmable devices: Systems that need high bandwidth, high density, high performance, design flexibility and low cost. The key features of this chip are presented below.

Processing System (PS)

The main components of the PS are the application processing unit (APU) and the real-time processing unit (RPU). The former is either a quad-core or dual-core with CPU frequency of up to 1.5 GHz. It can be operated in 64-bit or 32-bit operating modes. The RPU is dual-core with a CPU frequency of up to 600 MHz. Other components of the PS are listed below:

- Graphical Processing Unit (GPU): The MPSoC contains the Arm Mali-400 based GPU with a frequency of up to 667MHz. It supports OpenGL ES 1.1

and 2.0; and OpenVG 1.1.

- External Memory Interfaces: The MPSoC provides 32-bit or 64-bit interfaces to DDR4, DDR3, DDR3L, or LPDDR3 memories, and 32-bit interface to LPDDR4 memory. It also provides static memory interfaces such as eMMC, NAND flash or SPI serial NOR flash.
- On-chip Memory: The UltraScale+ MPSoC device has the following on-chip memory resources
  - 256KB on-chip RAM in PS
  - Up to 36Mb on-chip RAM (UltraRAM) in PL
  - Up to 35Mb on-chip RAM (block RAM) in PL
  - Up to 11Mb on-chip RAM (distributed RAM) in PL
- Dedicated I/O Peripherals and Interfaces: Some of the on-board peripheral controllers and interfaces on the PS include
  - PCI Express
  - SATA Host: supports up to two channels and data rates of 1.5, 3.0, and 6.0 Gb/s
  - DisplayPort Controller: supports up to two TX lanes and data rates of up to 5.4Gb/s
  - Four 10/100/1000 Mbit Itri-speed Ethernet media access controller (MAC) peripherals with IEEE Std 802.3 and IEEE Std 1588
  - Two USB 3.0/2.0 Device, Host, or OTG peripherals, each supporting up to 12 endpoints
  - Two SD/SDIO 2.0/eMMC4.51 compliant controllers

  Other interfaces include UARTs, SPIs, I2C and multiplexed I/O (MIO)
- Interconnect: There is need for high-bandwidth connectivity within the PS and between PS and PL. Xilinx adopted the Advanced eXtensible Interface (AXI) protocol as the main interconnect in their devices. AXI is part of ARM AMBA, which is a family of micro controller buses. The AXI4 is the second major version of AXI, released in 2010 and has three types:

  - AXI4: For high-performance memory-mapped requirements
  - AXI4-Lite: For simple, low-throughput memory-mapped communication
  - AXI4-Stream: For high-speed streaming data

Other components of PS include the System Memory Management units; Platform Management Unit; Configuration and Security Unit; and System Monitor in PS.

Programmable Logic (PL)

The Zynq Ultrascale+ MPSoC contains a PL with the following main components:

- Configurable Logic Blocks (CLB): Look-up tables (LUT), Flip-flops and cascadable adders.
- 36Kb Block RAM: True dual port with up to 72 bits wide and can be configured as dual 18Kb.
- UltraRAM: 288Kb dual port of 72 bits width and with error checking and correction.
- DSP Blocks: These include 27 x 18 signed multiply, 48-bit adder/accumulator and 27-bit pre-adder.
- Programmable I/O Blocks: These are from 1.0V to 3.3V I/O that support LVCMOS, LVDS, and SSTL.
- PCI Express: There are up to 5 integrated blocks in select devices that support up to Gen3 speeds.

Other components include 100G Ethernet MAC/PCS, Video Encoder/Decoder (VCU) and System Monitor in PL for on-chip voltage and temperature sensing.

Ultra96v2 Evaluation Board

MPSoC (Multi-Processor SoC) implies that the processing system is comprised of multiple (and heterogeneous) processors. The Zynq UltraScale+TM MPSoC has three distinct variants (CG, EG and EV) based on the type and range of processors in its PS. All the three variants have a common real-time processor (RPU) which is the Dual-core Arm Cortex-R5F processor. The difference is based on how many

cores is the Application Processing Unit (APU), whether GPU is included and finally if there is a video codec. The differences are highlighted in Table 4.

Table 4: Zynq MPSoC Devices

Feature	**CG Devices**	**EG Devices**	**EV Devices**
RPU	Dual-core Arm Cortex-R5F	Dual-core Arm Cortex-R5F	Dual-core Arm Cortex-R5F
APU	Dual-core Arm Cortex-A53	Quad-core Arm Cortex-A53	Quad-core Arm Cortex-A53
GPU	NA	Mali-400MP2	Mali-400MP2
VCU	NA	NA	H.264/H.265

The Ultra96v2 is based on the EG MPSoC variant. The Zynq device utilized is the ZU3EG A484 MPSoC. In addition to the Xilinx Zynq MPSoC device, the Avnet Ultra96 board has the following peripherals and components that offer greater accessibility and utility in the evaluation phase:

- LPDDR4 Memory: Micron 2 GB (512M x 32)
- Microchip Wi-Fi / Bluetooth
- Mini DisplayPort (MiniDP), USB 3.0 and USB 2.0 interfaces
- I/O headers

Figures 3 and 4 show the Ultra96v2 Board and a block diagram of its features.

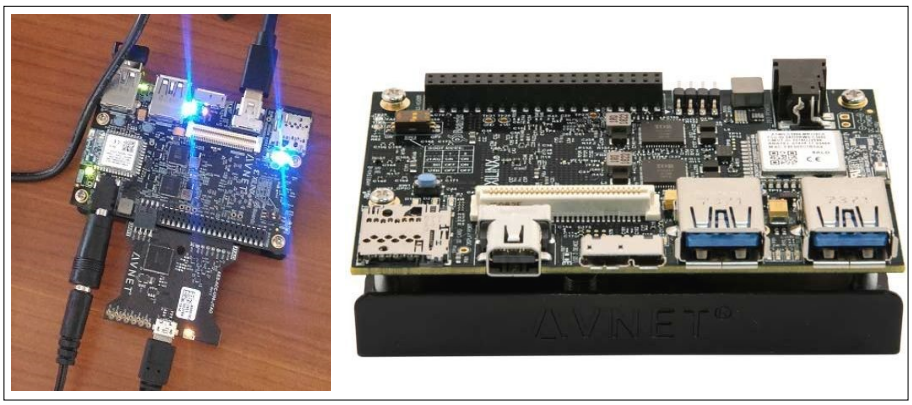


Figure 3: Avnet Ultra96v2 Board

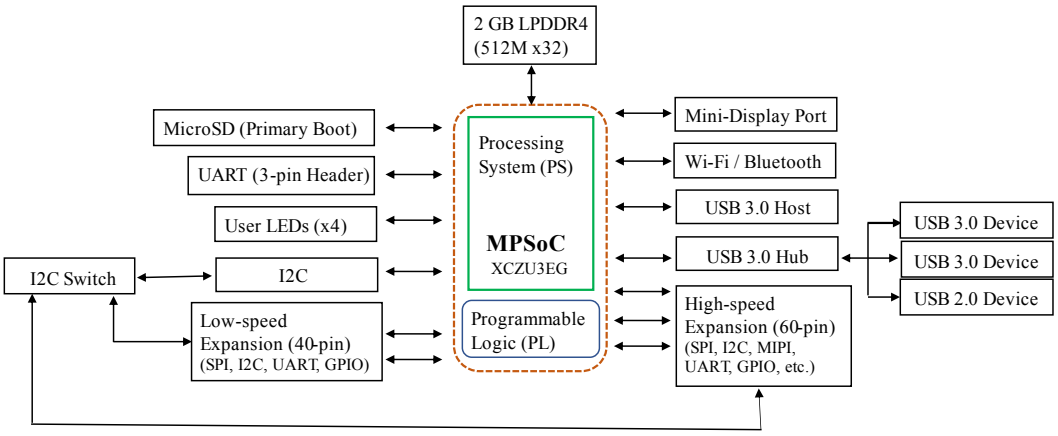


Figure 4: Ultra96v2 Board Block Diagram

3.1.1.3 Radiation Tolerant Kintex UltraScale FPGA

The devices that have been covered above are not optimized for space operation. In May 2020, Xilinx unveiled a radiation-hardened FPGA for in-orbit and space application. The RT Kintex device was a response of Xilinx to the current trends in space activities. In the coming years, most of the satellites launched are predicted to be smaller spacecrafts targeting the lucrative space-based Internet and Earth Observation data-analytics markets [10]. These will require high-throughput, low-latency communication. More so, such satellites are typically in the LEO with a lifespan of 3 to 5 years and constrained by power consumption and cost. With these constrains and performance expectations, FPGAs are a compelling candidate for onboard operations. The RT Kintex offers the following advantages in space applications:

Radiation-effects mitigation and hardness: This is due to the CMOS scaling of planar transistor technology that has made the device less prone to total-dose and latch-up effects. Heavy-ion and total-dose tests have been carried out that showed estimated BRAM sensitivity to be 4.7E-7 upsets/bit/day for LEO and 2E-8 upsets/bit/day for GEO.

In-orbit reconfigurability: The device is SRAM-based and has no limit on the number of times it can be reconfigured both in the lab and in orbit. Some key

advantages of in-orbit reconfigurability are presented below:

- It offers maximum flexibility allowing new and better communication standards to be uploaded to improve system performance
- Enables multiple experiments to be prototyped from a single payload for technology demonstrator satellites.
- On-board re-programming reduces the size of the payload hardware, enabling the use of various algorithms to improve applications, system security, and fault tolerance.
- Partial reconfiguration allows modification of a specific active region implemented within the FPGA without compromising the integrity of the applications running elsewhere within the device that use the imported logic.

Flexible, Digital-Beamforming Telecommunication Satellites: These communication satellites are increasingly using phased-array antennas and digital beamforming techniques to combine multiple individual antenna elements to improve overall performance, increase gain, cancel out interference, and steer the array so it is most sensitive in a particular direction. This allows operators to change and optimize reception and transmission in response to changing link requirements in real-time. Re-configurability of the device also allows DSP slices to adapt to different carrier and de-modulation standards.

Real-time streaming of earth observation, remote sensing video: In the Space 2.0 , there will be need for real-time 4K UHD and 8K super high-resolution streaming video to enable novel remote-sensing applications. This will enable services such as:

- High-resolution, streaming video SAR and LIDAR
- On-board processing to detect and identify moving targets in real-time
- Calculation of the moving targets' velocities
- Advanced tracking, intelligence, surveillance, and reconnaissance

The device offers huge I/O and memory bandwidths together with the on-board processing capability to enable future Earth Observation applications.

On-board AI for Autonomous Space Exploration:

- Future space exploration will require significantly more on-board processing to stream images back to Earth in super high-resolution.
- Landers and rovers have to manage local environmental conditions, and future robots will use AI to explore, map, and navigate terrain to avoid hazards, pick-up objects, and collect and analyze samples.
- Future robotic subsystems will also require significant increase in image processing, autonomous navigation, reading telemetry data from sensors, and controlling actuators. All of this information will have to be processed in real-time to enable remote space exploration.

The device offers huge I/O and memory bandwidth together with the on-board processing capability to enable the next generation of remote and autonomous space exploration. NASA's Spirit and Opportunity rovers used Xilinx's Virtex-4QV FPGA to explore the Martian surface whilst the V5QV space-grade device was used for off-line processing on the MARS2020 mission.

A summary comparison of the three devices presented in the preceding sections is presented in Table 5.

Table 5: Comparison of RT Kintex, Kintex-7 & Zynq MPSoC

Feature	**Kintex-7 XC7K325T**	**RT Kintex XQRKU060**	**Zynq MPSoC ZU3EG A484**
Evaluation Board	KC705	NA	Ultra96v2
System Logic Cells	326,080	725,550	154,350
CLB Flip-Flops	407,600	663,360	141,120
CLB LUTs	203,800	331,680	70,560
DSP Slices	840	2760	360
Block RAM Blocks	890	1080	216
Block RAM (Mb)	16	38	7.6
Maximum Distributed RAM (Mb)	4	9	1.8

3.1.2 Xilinx Development Tools

3.1.2.1 Vivado Design Suite

This is a Xilinx IDE that provides the environment and tools to design a custom architecture on the programmable logic. It supports most of the Xilinx devices that include FPGAs and SoCs. The design suite contains Xilinx intellectual properties (IPs) that facilitate the hardware design. It also allows creation of user IPs and use of 3rd party IPs. There are several ways in which Vivado is used for hardware design. These will be briefly expounded below:

Register Transfer Level Flow:

Vivado has traditionally supported hardware design at the RTL level. This is an abstraction layer that creates high-level representations of a digital circuit. This layer is mostly used in Hardware Description Languages (HDL). Vivado supports the two major HDLs i.e. Verilog and VHDL. These languages allow a high-level description of the desired logic design in a human-friendly language. This design is then interpreted by a logic synthesis tool, which is incorporated into Vivado. The synthesis step converts the high-level hardware description into a gate level description. The interconnections amongst these gates is then performed by the placement and routing tool, which is also embedded into Vivado. This creates the full physical layout of the hardware design on the programmable logic. This design process is shown in Figure 5.

High Level Synthesis:

The Vivado HLx editions enable the use of C/C++ for high-level hardware descriptions. This is a higher level of abstraction compared to the RTL / HDL level. It contains extensive libraries for including built-in support for arbitrary precision data types, streams and vectorized data types. These libraries include:

- ap_int.h - This enables arbitrary precision data types including integer and fixed-point.
- hls_stream.h – This contains the models for streaming data structures.
- hls_vector.h Enables vectorized types and operations

- HLS Math – This has extensive support for the synthesis of the standard C (math.h) and C++ (cmath.h) math libraries. The support includes floating point and fixed-point functions such as tanf, sinhf, cosf, floorf, rsqrt etc.

The C code is first validated to ensure its correctness. This validation is performed using a test bench code that is also in C. After this, the C design is synthesized into an RTL design and the previous steps for RTL design are followed as shown in Figure 5.

Block-based IP Integration:
The Vivado IP Integrator provides a graphical and Tcl-based development flow for IPs. It provides an abstraction at the interface level as opposed to signal level when working with IPs. This allows designers to make connections between IPs. The interconnections are mostly done using the standard AXI4 interface, although the Integrator supports other interfaces. It supports intelligent auto-connection of key IP interfaces, one-click IP subsystem generation, real-time DRCs, and interface change propagation. The IP Integrator is advantageous in that it enables seamless inclusion of hierarchical subsystems into the overall design. It also supports both processor and processor-less designs including combination of DSP, video, analog, embedded, connectivity, and logic. It also facilitates recognition and correction of common design errors and performs automatic IP parameter propagation to interconnected IP. The IP-based block design flow is illustrated in Figure 5.

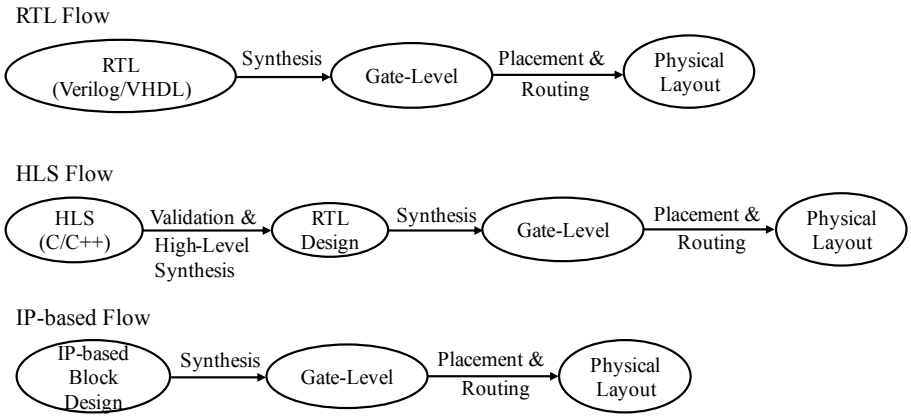


Figure 5: Vivado Design Flows

3.1.2.2 Vitis Unified Software Platform

In 2019, Xilinx unveiled the Vitis platform that aimed to form the base for all development on Xilinx devices. The main features of this unified platform are enumerated below:

- Enables development of embedded software and accelerated applications on Xilinx platforms such as FPGAs, SoCs and Alveo accelerator cards.
- Facilitates development at different levels of abstraction. It enables integration with high-level languages and frameworks such as C, C++, Python, Tensorflow and Caffe while also enabling RTL-based design flows and providing low-level runtime APIs.
- It offers both GUI and command line development tools.

The main components of this unified platform are briefly presented in the following sections.

Vitis Core Development Kit:

This offers the complete set of graphical and command-line tools including Vitis compilers, analyzers and debuggers targeting algorithms developed in C, C++ or OpenCL. The kit also bundles the Vivado design suite.

Vitis Accelerated Libraries:

These are extensive sets of open-source, performance-optimized libraries that offer out-of-the-box acceleration with little code modification. Common libraries such as those for DSP, Statistics, Math and Linear Algebra offer core functionality for a wide range of diverse applications. There are also domain-specific libraries for applications such as Data Analytics, Data Compression, Vision and Image Processing and others.

Vitis AI Development Environment:

This offers an optimized development environment for accelerating AI inference on Xilinx devices and platforms. It supports the main deep learning frameworks such as TensorFlow and Caffe. The Vitis AI environment offers comprehensive APIs to

prune, quantize, optimize, and compile custom trained networks for inference on Xilinx devices such as FPGAs, SoCs and Alveo accelerator cards.

Xilinx Runtime Library (XRT):
This is a core component of the Vitis Unified Software Platform and Vitis AI Development Environment. It provides a communication interface between the application code and the accelerated kernels deployed on the programmable logic of the Xilinx devices such as Zynq UltraScale+ MPSoC based embedded platforms and Alveo accelerator cards. It runs on the host CPU i.e. on the processing system. The key benefits of the XRT are enumerated below:

- Allow developers to focus on application development by abstracting away hardware level of the Xilinx platforms. Hence, they can develop applications in high level languages and frameworks such as C/C++, Python, TensorFlow and Caffe.
- The XRT offers common API across various Xilinx platforms. Hence developers can design and seamlessly port accelerated applications for Edge, On-premise and Cloud deployments.
- It enables developers to leverage Dynamic Function eXchange (DFX) that allows reconfiguration of modules within an active design. This allows them to easily swap out different accelerator binaries on Xilinx platforms without the bottleneck of low-level details.

3.1.2.3 Xilinx Software Development Kit (XSDK)

This is the Integrated Design Environment for creating embedded applications for Xilinx platforms such as SoCs, MPSoCs and softcore processors such as MicroBlaze. The IDE directly interfaces to the Vivado embedded hardware design environment. It contains a full suite of libraries and device drivers. Its key features are briefly presented in the following sections.

Drivers and Libraries:
The XSDK includes many libraries and drivers that are user customizable. These support all Xilinx hardware IPs, kernel library and networking and file handling

libraries. They can scale for the custom embedded design based on various requirements such as feature needs, memory, and hardware capabilities.

System Debugger:
This is integrated into the IDE and supports all the common debug features such as setting breakpoints, viewing contents of the memory, and stepping through program execution. It can also simultaneously debug programs running on different processors in MPSoC systems. The system debugger also supports cross-triggering between processors in the PS and processors and hardware in the PL.

Custom Design Aware:
From the Vivado hardware design, the SDK is able to automatically understand the embedded design and auto-configure several key parameters. These include memory maps, peripheral register settings, tools and library paths, Linux and bare-metal Board Support Packages (BSPs). This auto-generation of critical system software and custom design-aware pre-configuration facilitates faster software development.

System Performance Analysis and Optimization:
The XSDK provides a toolbox for modelling, measuring, analyzing, and optimizing the system design. This allows IP blocks that will be in the programmable logic to be modeled even before they are completed. Hence they can be analyzed and an approach on system optimization can be made, for example, by optimizing the software part, splitting the function between the processor and programmable logic, or by migrating the whole function to programmable logic. Since 2019, the SDK has been bundled into the Vitis unified software platform.

3.1.2.4 PetaLinux
This is a tool provided by Xilinx for customizing, building, and deploying Embedded Linux solutions on the Xilinx processing systems. The target devices include Zynq UltraScale+ MPSoC, Zynq-7000 SoCs and MicroBlaze softcore processor. On the host, the PetaLinux tools are composed of the following:

- Command-line interfaces

- Bootable system Image builder
- GCC tools and debug agents
- Integrated QEMU full system simulator
- Application, device driver and library generators and development templates

These tools can largely be categorized into three main functions:

Custom BSP Generation Tools:
With the underlying hardware design exported from Vivado, the PetaLinux tools automatically generate a Board Support Package (BSP) for the custom design. This includes device drivers for the processing IP cores, kernel and bootloader configurations. Hence the tools enable synchronization between the software platform and the hardware design as it develops and incorporates new features and devices.

Linux Configuration Tools:
The PetaLinux tools enable the developer to customize the boot loader, Linux kernel, or Linux applications. They also allow customization of the file system, libraries, and system parameters. Using QEMU, the Linux build can be booted and tested on a full system simulator.

Software Development Tools:
Once the product's software baseline (BSP, device drivers, core applications) has been created, the PetaLinux tools enable developers to package and distribute all software components for easy installation and use across PetaLinux developers. The PetaLinux tools also provide quick-start Linux Images that include pre-configured boot loaders, system images and bitstream. This provides a platform that is ready for application, library, and driver development.

3.1.3 Hardware Description Languages

The programmable logic of an FPGA needs a different programming approach from the one utilized in software development. The end result of the FPGA programming is a physical mapping of its logic circuitry whilst for the software program-

ming it is a lower level machine code that runs on an already hard-wired logic circuitry. Therefore, the characteristics and features of HDL are different from those of a programming language. The major operational difference is that programming languages mainly describe sequential operations whilst hardware description languages allow various configurations at the hardware level. For example, they allow concurrent operations whereby separate parts of the digital logic operate simultaneously. HDL also incorporate features that allow optimum logic configuration at the hardware level. These include propagation delay and timing information of the various connections in the digital circuit. The two main HDLs are Verilog and VHDL. These are briefly described in the following sections.

3.1.3.1 Verilog

This was initially developed as a proprietary HDL around 1984 by Gateway Design Automation Inc. In December 1995, it became an IEEE standard [57]. Verilog is aesthetically similar to C programming language. Functionally, Verilog allows hardware description at different levels of abstractions as briefly presented in the following sections.

Behavior Level: This is used to model a test bench. Hence it describes the digital circuit at a high-level and is not synthesizable i.e. will not be converted into a circuit mapping. This allows verification of a design before proceeding to real hardware implementation.

Register Transfer Level: This is the beginning of describing a design that is synthesizable. The RTL code describes the digital circuit design by operations and the transfer of data between the registers. It contains synthesizable syntax, clocking and timing bounds information.

Gate Level: This is generated from the RTL code as a result of the synthesis step. This code describes the system at the discrete signal level i.e. gate level. Hence, these signals can only have definite logical values (0, 1, X, Z) and the operations are predefined logic primitives such as gates NOT, AND, OR, XOR etc.

The basic unit of Verilog modeling is the *module*. It contains the RTL code that describes the system design using data types, operators, assignments, expressions, clocking information and other parameters required for a complete digital circuit design. As described above, Verilog supports four logic values: 0, 1, X, Z. Verilog has two main data types. These are *register* and *net* data types. With these, signals and variables can be declared at the RTL level in the Verilog design. The *register* type is declared using *reg* and is used for variables that require some form of storage or persistence. Hence, it is used for latch or flip-flop where it stores values until the next event occurs. Other less commonly used *reg* types include *real*, *integer*, *time* and *realtime*. The *net* type is declared mostly using *wire* and is used for combinational logic and intermediate signals for links. In addition, there are other less commonly used net types such as *wor*, *wand*, *tri* etc. Inside the module, the RTL code is contained mainly in two types of expression statements. These are the *initial* and *always* statements, that are both used to generate events. The *initial* statement is used to declare an initial value or to describe behavioral level. It is not synthesized into a digital circuit and runs only once when the module is activated. On the other hand, the *always* statement is used to describe synthesizable digital circuit. This includes sequential logic such as flip-flops and complex combinational logic. The *always* statement runs every time a specified event occurs. Inside these statements, the system design is modeled via various assignments of variables and signals. There are two types of assignment statements: *blocking* and *non-blocking*. These operate similarly for *net* data types but differently for *register* types. In the latter, the data transition in the *blocking* assignment occurs sequentially whilst it occurs simultaneously in the *non-blocking* assignments [58].

3.1.3.2 VHDL

This stands for VHSIC (Very High Speed Integrated Circuits) HDL. Its development was initially sponsored by US DoD in the 80s and was transferred and ratified as an IEEE standard in 1987. Similar to Verilog, VHDL offers various levels of design. The highest level of abstraction is the behavioral level. It is mostly used in design of test benches by describing the relationship between the input and output signals. VHDL is more verbose than Verilog and is strongly typed. For

this reason, it provides more features for high-level hardware modeling.

The RTL code based on VHDL is composed of various design units. These include *entity*, *architecture*, *packages* etc. The entity is the basic design unit, that is at the top level of every system design. It defines the interface to the external world i.e. the hardware design. This is by defining the input and output signals or ports. The internal structure and behavior of an entity is defined by an *Architecture*. This specifies how the hardware circuit works and describes its implementation. An entity can contain one or more architectures. The circuit implementation description by the architecture can be structural, behavioral, or functional (dataflow). The architecture declares the components (gates), signals, constants, and subprograms. Each of the subprogram or process executes concurrently with respect to the other subprograms. However, the statements within the process execute in sequential order. There are two kinds of subprograms: procedures and functions. Both perform sequential computations. However, a procedure returns values in global objects or by storing values into formal parameters whilst a function returns a value as the value of the function. The function doesn't change its formal parameters.

Unlike Verilog, VHDL supports many data types as well as user defined ones. The predefined data types include *numeric* (real or integer), *boolean*, *bit*, *bit-vector*,*character*, *string*, and *time*. VHDL also contains libraries and packages, while there is no package definition in Verilog. A VHDL package contains the declarations of commonly used and shared objects, data types, functions, procedures, and components. Hence, subprograms, data types and other features declared in the package can be used in different entities and architectures. A VHDL design uses configuration statements to associate the exact entities with desired component instances in the design. This allows the configuration of different designs. For example, one configuration can result into a test bench for functional simulation whilst another configuration can create a synthesizable logic design. Another difference between VHDL and Verilog is that the latter is case-sensitive whilst the former is not. In Verilog, to use a component instance, you just need to instantiate it in the module with a correct port map. However, in VHDL the component first needs to be declared in the architecture or in the package. Verilog contains compiler directives whilst VHDL does not [59].

3.2 Convolutional Neural Networks

3.2.1 Multi-layer Perceptron Neural Networks

The traditional multi-layer neural networks (MLP) were ineffective in dealing with large data sets especially in the computer vision and image processing fields. A simple MLP would require many parameters for the processing of a single image. Hence one would require a lot of data to prevent overfitting and consequently the computational requirements would be very high. This hindered their scaling to applications in the field. They also suffered from vanishing gradients as the layers increased. The convolutional networks were an improvement to the MLP networks. Figure 6 shows the general architectures of MLP and CNN networks.

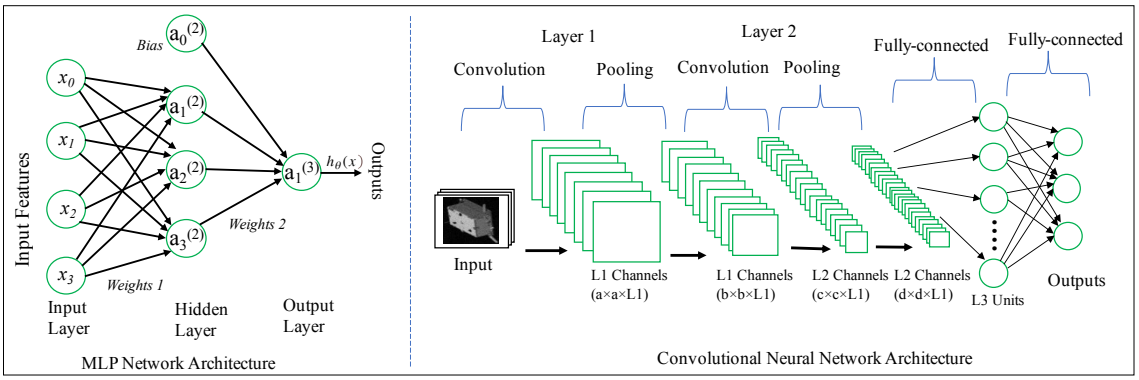


Figure 6: MLP vs CNN Architectures

In MLP, the output of a neuron output, for example one in the hidden layer, is given by Equation (1).

$$h_\theta(x) = g(\theta^1_{10}x_0 + \theta^1_{11}x_1 + \theta^1_{12}x_2 + \theta^1_{13}x_3) = g(u) \tag{1}$$

Where g is the activation function, x are the input features and θ are the weights associated with each neuron connection.

3.2.1.1 Activation Function

The activation function defines the output signal from the neuron in terms of its net input signal u. The commonly used activation functions in MLP are shown in Figure 7 and briefly discussed in the following sections.

Binary Threshold Function:

This is also called **Heaviside Function**. It is the simplest form of activation function. The output of the neuron can only take on two values. If the input is above a certain threshold, the output changes from one value to another, but otherwise remains constant as shown in Equation (2). [60].

$$y = g(u) = \begin{cases} -1, \ u < 0 \\ 1, \ u \geq 0 \end{cases} \tag{2}$$

The function is not differentiable at the threshold input value e.g., at u=0. More so, the derivative is 0 for all the other input values. This property makes the Heaviside function unfavorable since in most cases, it will be important to obtain such derivatives in the tuning of the parameters (weights). For example, back propagation algorithm requires the partial derivatives of a cost function with respect to each of the weights.

Hyperbolic Tangent Function:

This is a differentiable function (smooth curve). The output value, y, takes values in the range of *(-1, 1)*. Strongly negative inputs to the hyperbolic tangent function will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. These properties make the network less likely to get "stuck" during training.

$$y = g(u) = \tanh u \tag{3}$$

This function is fully differentiable, with its first derivative given as:

$$g^{'}(u) = 1 - \tanh^2 u \tag{4}$$

Sigmoid Function:

This is also called the **logistic function**. It has a nice biological interpretation and is the most used activation function in neural networks. It is a monotonous crescent function which exhibits a good balance between a linear and a non-linear behavior. The sigmoid function crosses 0.5 at the origin and then flattens out with asymptotes at 0 and 1. It maps any real number to the (0, 1) interval, making it

useful for transforming an arbitrary-valued function into a function better suited for classification. This function is given in Equation (5).

$$y = g(u) = \frac{1}{1 + e^{-u}} \tag{5}$$

The logistic sigmoid has a downside in that it can cause a neural network to get "stuck" during training. This is due in part to the fact that if a strongly negative input is provided to the logistic sigmoid, it outputs values very near zero. This can result in model parameters (weights) that are updated less regularly during the training phase and are thus "stuck" in their current state. However, this is easily overcome by effective feature (input) scaling and advanced optimization techniques. The scaling refers to instances where the inputs have very different scales (e.g., temperature and volume of a liquid) and thus these inputs are scaled (standardized) to comparable range of values.

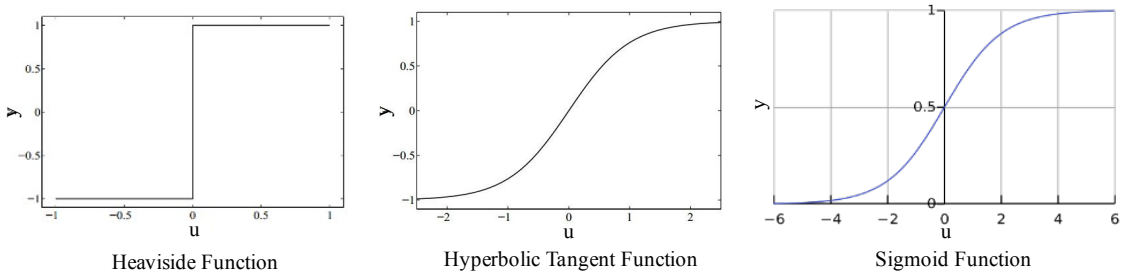


Figure 7: Activation Functions

3.2.2 CNN Networks

In convolutional networks, a matrix, known as a filter or kernel, operates on the inputs of the layer and the kernel output forms the input to the next layer. Each element in the output is obtained by summation of the element-wise product of the input matrix and the filter. In computer vision, filters are commonly used in edge detection operations. Such kernels are convolved with the image for edge detection. All the above kernels are chosen a priori i.e., the matrices are hard coded. In CNN, these kernels are not hardcoded. The algorithm learns them during the training phase. Hence, their values are parameters that are learned via backpropagation at training. Examples of such kernels are shown in Figure 8

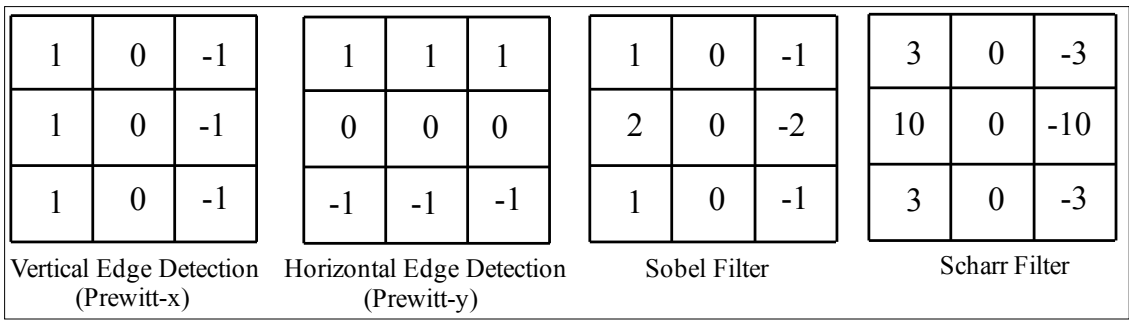


Figure 8: CNN Filters

There are various types of layers in a convolutional network. These include convolution, pooling and fully connected. This section will briefly present background information on the layers.

3.2.2.1 Convolution Layers (CONV)

These are the fundamental building blocks of a CNN architecture. They are composed of the kernels introduced above. The basic convolution operation of an $m \times m$ image with a $k \times k$ kernel will result in an output of $m-k+1 \times m-k+1$. This has a negative impact in that after every convolution operation, the image shrinks. Due to the numerous layers in CNNs, this will lead to a minuscule image after few layers. This can be addressed by use of *padding*.

Padding is the addition of extra borders to an image before performing the convolution. For example, this border can be an extra pixel all around the image edges. The concept of padding results into two types of convolutions: *valid* and *same* convolutions. In valid convolution, there is no padding utilized. In same convolution, the padding is such that the output has the same size as the input. With padding, the output of the convolution is $m-k+2p+1 \times m-k+2p+1$, where p denotes the number of extra pixels added on each image edge.

In practice, convolutions are performed over volume with multiple filters. For example, in deep learning the images are mostly RGB. These 3 channels are convolved simultaneously. Hence each image channel will require a filter to convolve with. Consequently, 3 filters are needed for RGB images. In general, the number

of channels, nc, in the filter should be equal to those in the input image. In addition, multiple filters are needed in a typical convolution operation. This is because there is need to detect multiple features in the image concurrently. Thus, typical convolution operations in deep learning detect multiple features of multi-channel images. Figure 9 shows the above convolutions flow and the notations in a typical CNN architecture.

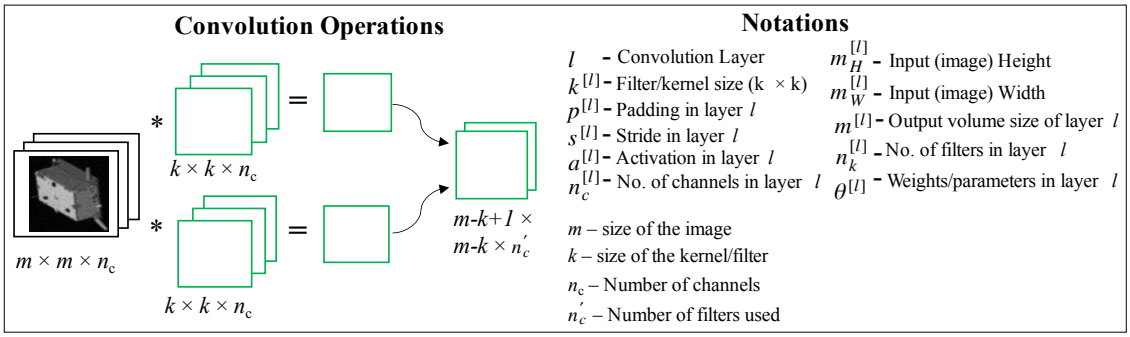


Figure 9: Convolution Flow and CNN Notations

With the notation shown in Figure 9, the activations (outputs) of a convolution layer and the inputs (activations from previous layer) are given by Equation (6).

$$\begin{aligned} a^{[l]} &= m_H^{[l]} \times m_W^{[l]} \times n_c^{[l]} \\ a^{[l-1]} &= m_H^{[l-1]} \times m_W^{[l-1]} \times n_c^{[l-1]} \end{aligned} \tag{6}$$

The output volume size and the total number of weights in a single convolution step (layer) are given by Equation (7).

$$\begin{aligned} m^{[l]} &= \frac{m^{[l-1]} + 2p^{[l]} - k^{[l]}}{s^{[l]}} + 1 \\ \theta^{[l]} &= k^{[l]} \times k^{[l]} \times n_c^{[l-1]} \times n_k^{[l]} \end{aligned} \tag{7}$$

3.2.2.2 Pooling Layers (POOL)

As shown by the CNN architecture in Figure 6, a layer is composed of a *convolution* operation and a *pooling* operation. A convolution results in a feature map. This is a grid of features that have been learnt and derived from the input image or features. The input and output feature maps have a high correlation; hence the

output feature map has high sensitivity to a change in the location of a feature in the input feature map. This makes it difficult for the CNN to generalize to different input features e.g., images of different objects and backgrounds. To address this, a pooling layer is utilized. The pooling layer down samples the output feature maps, making them invariant to local translation, i.e., less sensitive to changes in the location of features in the input map. It achieves this by summarizing features in a given region of the output map. Hence the output of the pooling layer has fewer dimensions than the output of the convolution layer. This down sampled feature map forms the input of the next layer. This reduces the number of parameters to learn and the computational requirements of the network.

The pooling layer can be implemented in different ways. Two of the main types of pooling layers is the *max pooling* and the *average pooling*. Max pooling picks the maximum element in the block or region of the feature map that is covered by the pooling filter. This has the interpretation that the most prominent feature detected is preserved as the output of the pooling filter and consequently as input to the next layer. The hyperparameters of the pooling filter are mainly the size of the filter, k, and the number of strides, s. Padding is rarely used in pooling.These are fixed parameters and hence are not learned in the training phase. The output of a max pooling is given by the same equation of output volume size in Equation (7).

Average pooling, on the other hand, picks the average of the elements in the region of the feature map covered by the pooling filter as its output and consequently as the input to the next layer. It has similar hyperparameters to that of the max pooling and its output is similarly given by equation. Pooling can further be categorized as either local or global. Local pooling refers to pooling acting on a small region or cluster of the feature map whilst global pooling acts on the whole feature map. Hence, in the latter, each channel in the feature map is reduced to a single value either via max pooling or average pooling.

3.3 Spacecraft Pose Estimation

Pose determination/estimation is the capability of an active spacecraft (chaser) to accurately estimate its relative position and attitude (orientation) with respect to an active or inactive target in close-proximity in space. The position is usually in Cartesian coordinates and it can be expressed as a position vector as in Equation (8), where $\hat{a}_x$, $\hat{a}_y$ and $\hat{a}_z$ are the base vectors in the Cartesian plane.

$$\overline{r} = x\hat{a}_x + y\hat{a}_y + z\hat{a}_z \tag{8}$$

On the other hand, orientation in three-dimensional (3D) can be represented in various ways such as use of Euler angles, rotational matrices and quaternions. In Euler angles, an orientation can be represented with 3 numbers (Euler angles) in 12 possible Euler angle sequences such as *xyz*, *xzy*, *zxy* etc. One shortcoming of Euler angle approach is that they suffer from gimbal lock whereby two axes effectively line up leading to loss of a degree of freedom. They also require extensive trigonometry operations when converting between different rotational matrices [61]. Quaternions offer a more effective way of representing 3D orientations [62, 63]. They are four-dimensional with one real component and three components in the *ijk* imaginary space, as expressed in Equation (9).

$$q = q_0 + q_1 i + q_2 j + q_3 k \tag{9}$$

where *i,j,k* satisfy the conditions in Equation (10)

$$\begin{cases} i^2 = j^2 = k^2 = ijk = -1 \\ ij = k,\ jk = i,\ ki = j \\ ji = -k,\ kj = -i,\ ik = -j \end{cases} \tag{10}$$

Whereas, rotations in Euler Angles are obtained by specific angle sequences, quaternions offer an analogous single rotation around a unique axis that results in the same rotation. The quaternion embeds this possibility i.e., angle and axis of rotation within its four-element vector. A rotation of a vector v in $\mathbb{R}^3$ to w by the quaternion q is given by:

$$w = qvq^* \tag{11}$$

The above equations represent the fundamental concepts of pose estimation using Cartesian coordinates and Quaternions.

There are two major categories in pose estimation techniques: cooperative and uncooperative spacecraft [64]. In the former, the target has inbuilt capacity to provide the chaser with information suitable for pose estimation. This capacity can be in the form of dedicated radio-link to interact with the chaser (active) or artificial markers that are easily recognized (passive). In uncooperative targets, there is minimal (known target) or no information (unknown targets) available to facilitate pose estimation. Spacecraft pose determination is achieved by relying mainly on electro-optical sensors, such as stereo cameras, monocular vision/infrared cameras, and Light Detection and Ranging (LIDAR) systems.

The advantages of the latter over the other two is that LIDAR systems are robust in poorly illuminated conditions and, hence, the target can be easily segmented from the background. They also offer very large operational ranges with constant accuracy levels. The main disadvantage is the hardware complexity and cost of such systems. Hence, monocular and stereo cameras are preferred for spacecraft pose determination. They have low power, mass, and cost requirements. However, they require additional algorithms to extract pose information, since they cannot provide direct measurements of the relative range [65].

These additional algorithms have been based on image processing techniques while using hand-engineered features. Consequently, the pose is estimated utilizing the target's image and its 3D model. However, such feature-based pose estimation is not scalable to spacecraft of different structural and physical properties. Improvements to such approaches have been proposed, with deep learning-based algorithms emerging as the preferred approaches. Such deep learning methods achieve pose estimation by two main ways. One approach is to discretize the pose space and solve the resulting classification problem. The other approach is to directly regress the relative pose from the input image. This becomes a regression problem for the deep learning network to solve.

CNN-based algorithms that use single monochrome images for pose estimation have been studied with satisfactory results. Hirano, in [66], demonstrates a CNN-based pose estimator for a spacecraft. The training data were obtained by use of a software simulator to generate synthesized images from a 3D model of the target

object. This pose estimator directly estimates the 3D keypoints of the model from which the pose is determined. Sharma, in [67], combines CNN with a Gauss–Newton algorithm. The CNN allows for feature detection without need for manual tuning of hyper-parameters whilst the Gauss-Newton algorithm provides the perspective equations for quantifying uncertainty in the estimated pose. Thaweerath et al. in [68] presented a CNN-based pose estimation for noncooperative docking operations. The position and orientation were predicted by directly regressing them from the input image.

4 Keypoints Localization in CNN-based Spacecraft Pose Estimation

As presented in 2.1.3, pose estimation is a fundamental task in various space applications and missions. Recent research has focused on how to utilize convolutional neural networks in vision-based pose estimation. This is following the success of such networks in terrestrial applications. In such approaches, the CNN is utilized in localizing the landmarks/keypoints on the spacecraft body. Once these landmarks have been identified by the CNN, further processing is performed to extract the pose information. In this chapter, the methodology followed in implementing the keypoints localization part of a CNN-based spacecraft pose estimation is presented. This includes an introduction to the dataset used in the training phase. The various data pre-processing tasks carried out are also presented, followed by an introduction to various CNN-based models that were investigated in this study. The results of all the approaches are presented in Chapter 7.

Methodology Goal

The typical CNN-based pose estimation flow is shown in Figure 10.

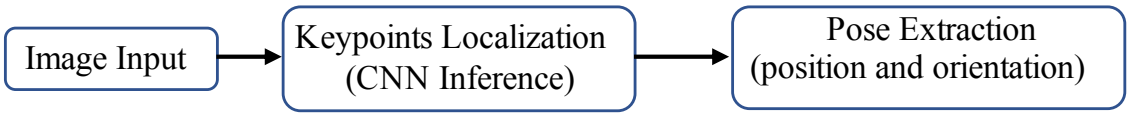


Figure 10: Typical CNN-based Pose Estimation Flow

However, the work in this thesis is focused on the CNN part only, which is the keypoints localization phase. The end goal is to implement this CNN part for inference on-board an FPGA. This is depicted in Figure 11.

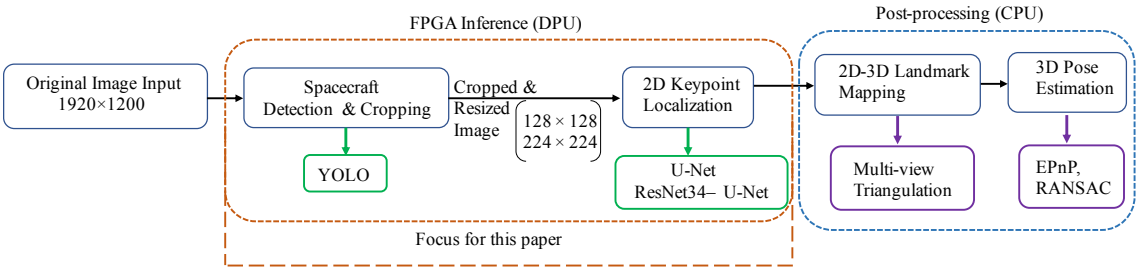


Figure 11: Focus of the Methodology and FPGA Inference.

Hence, the goal of this chapter is to present the training of the CNN part of a CNN-based pose estimation flow. This is focused on investigating the most appropriate CNN model with the highest accuracy in keypoint localization. Various network architectures are investigated and presented.

4.1 Dataset, Preprocessing and Training Environment

4.1.1 Dataset

In February 2019, the European Space Agency in collaboration with Stanford Rendezvous Laboratory launched a pose estimation challenge. They released the SPEED dataset that consisted of 12,000 synthetic images for training with an additional 2,998 for testing. The images are for a 3D rendition of a spacecraft. The 3D model of the Tango spacecraft was used in order to generate the synthetic images that were grouped into different categories representing different pose labels. The dataset also consists of 300 real images of the same spacecraft model and a few real images. Some of these images are shown in Fig. 12.

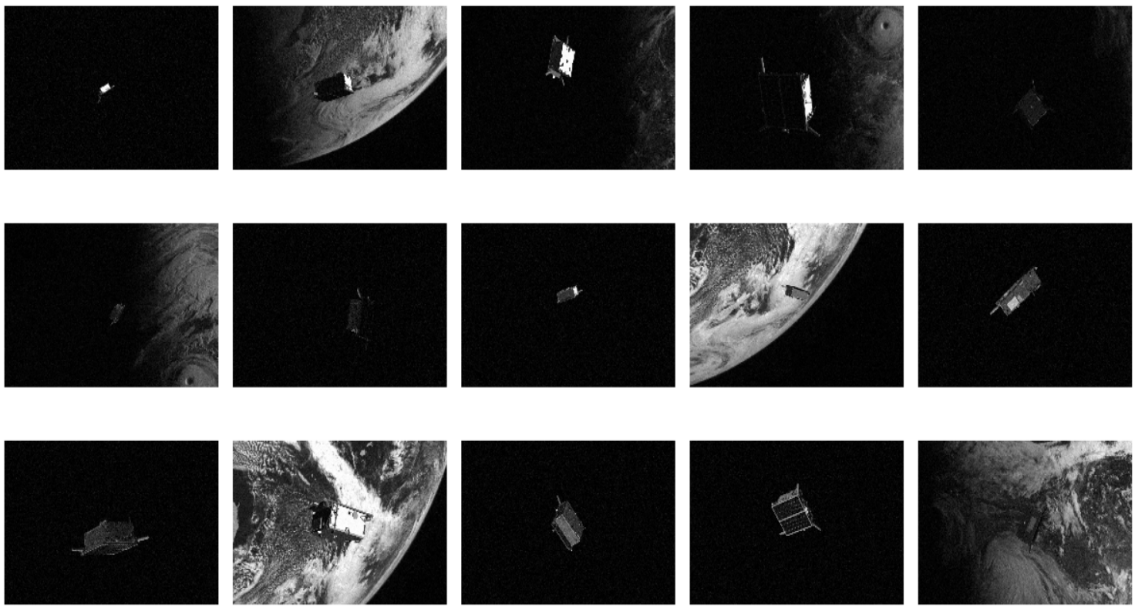


Figure 12: Images from SPEED Dataset

4.1.2 Preprocessing and Training Environment

The various networks investigated in this study required different data input architectures. Consequently, various preprocessing techniques were carried out on the SPEED dataset to prepare the inputs to these networks. Some of these preprocessing steps are presented in this section. The SPEED images data is originally in *json* files that contain the image index and training data. This includes data on the spacecraft pose and joints labels of select images for the training phase.

The first preprocessing step was to convert the data into a format that could be efficiently adopted for deep learning training. To accomplish this, Python scripts were written to efficiently handle the training data. The SPEED dataset has images with a pixel resolution of 1920 by 1200. This is a huge size for deep learning training. Hence, the first step was to directly resize the images appropriately. Some of the sizes used were 112, 128 and 224 pixels. The resizing of the images must be accompanied by corresponding resizing of the training data. The data was also converted to csv format. The scripts for these steps are included in the appendix. An example of the original dataset files and the pre-processed and formatted data for training is presented in Fig. 13.

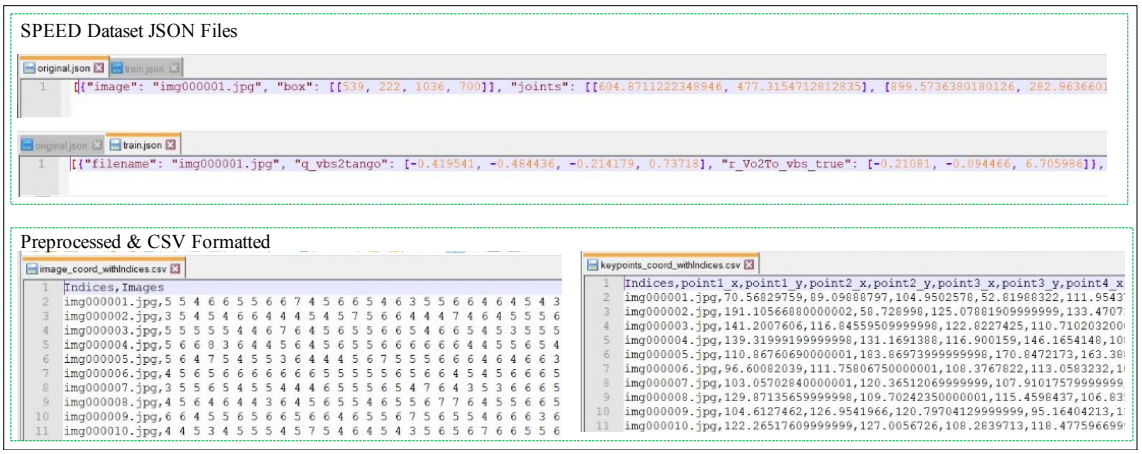


Figure 13: SPEED Json Files and Preprocessed CSV Files

Training Environment

Google Colab was used for training the various networks investigated in this work due to the availability of GPU allocations at no cost. For this, the images and

formatted data were uploaded on Google Drive for fast processing. Python scripts for various training regimes were drafted. These have been included in the appendix of this thesis report. Figure Fig. 14 shows some screenshots of the Colab training environment

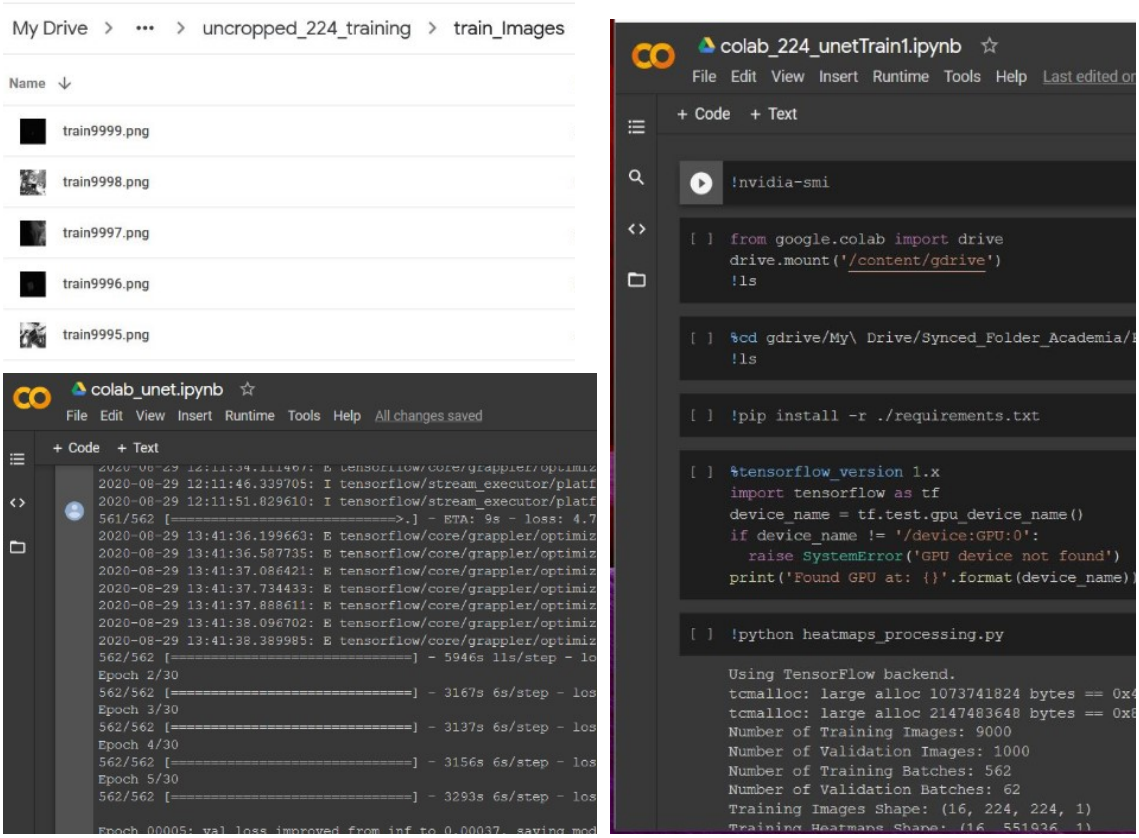


Figure 14: Google Colab Training Environment

The following sections present the different approaches and network architectures that were investigated in this work.

4.2 Regression vs. Detection-Based Approaches

In pose estimation problems, one of the key steps is the detection of pre-determined areas of the object, known as keypoints or joints. These are chosen apriori and when connected, they generally model the object and its orientation. Keypoints detection is therefore the determination of the precise pixel location of the target

joint. In spacecraft pose estimation, the illumination conditions of the spacecraft vary greatly in orbit. The images can be bright with the sun in the background or they can be dark when the spacecraft is in the Earth's shadow. The chosen keypoints are usually not all visible in the 2D image, some are occluded. Hence, the detection of the keypoints needs to be robust in both poor illumination conditions and occluded keypoints.

There are generally two major approaches to keypoints detection [69]. One is to directly regress the keypoint coordinates from an input image i.e., regression-based, whilst the other is to obtain coordinates from an intermediate heatmap i.e., detection-based. These two approaches make a tradeoff between the desirable traits of a deep learning model i.e., spatial generalization and end-to-end differentiability. The former is the ability of a network to generalize knowledge obtained at a particular location during the training phase to a different location during inference. The latter is a characteristic of the model to be composed of fully differentiable layers in an end-to-end manner that allows for backpropagation training.

In direct regression of keypoints, end-to-end differentiability is the driving factor. The input is the 2D image and the desired coordinates are the outputs. This approach has been used to varying success in human pose estimation problems, including the seminal work "DeepPose" by Toshev and Szegedy in [70]. The output layer predicts a pose/coordinate vector by minimizing a loss function that is usually the L_2 (mean square error) distance between the prediction and ground truth pose/coordinates. Hence, the network directly regresses the x, y coordinates of the keypoints.

In the detection-based approach, the output layer predicts a heatmap as opposed to coordinates. The loss function minimizes the error between the predicted and ground truth heatmaps. The coordinates are then extracted from the heatmaps in a post-processing step, which is usually to find the location p in the heatmap H_n with maximum likelihood for the keypoint K_n, as shown in Equation (12).

$$K_n = \underset{p}{\operatorname{argmax}}\, H_n(p) \tag{12}$$

where K_n is the n^{th} keypoint and H_n is the corresponding predicted heat map

for n^{th} keypoint. Because the end goal is the coordinates, this approach loses the end-to-end differentiability as this equation is not differentiable. Nevertheless, this detection-based approach is more effective than the regression-based approach since its training is supervised by dense pixel information, as will be demonstrated in this work. This approach was first utilized by Tompson et al. in [71]. The target heatmap was a 2D Gaussian with a small variance and mean centered at the ground-truth coordinate locations. This approach has been improved over the years, notably by the stacked hourglass network by Newell et al. in [72]. The main feature of the hourglass is the symmetry between the bottom-up processing (from high to low resolutions) and top-down processing (from low to high resolutions). For every layer on the way down, there is a corresponding layer going up. The full network is realized by stacking multiple hourglasses. The network output is a set of heatmaps that are a prediction of the probability of a joint's presence at every pixel. Consequently, smilar approaches have been used in works, such as [73, 74].

4.3 Network Model Architectures

The key criteria for selection of the network models in this work are based on their amenity to FPGA acceleration on the Zynq UltraScale+ MPSoC. Since the ultimate goal of this work was to investigate the acceleration of the CNN-part of pose estimation algorithms on FPGA, it is essential that the network layers and operations are supported by the DPU. More so, the operations need to be supported by the model quantization as well as compilation tools that are required for preparing the model for inference on Xilinx FPGAs. Table 6 shows operations that are supported by the Xilinx DPU and any imposed limitations.

Table 6: Operations Supported by Xilinx DPU.

Type	**Limitations**
Convolution	Kernel-width and kernel-height values (1 to 8)
ReLU	None
Pooling	2×2 and 3×3 Max Pooling
Concat	Concatenation in channel axis only
Elementwise	None
Inner Product	None

Three approaches were taken in this work as presented in the following sections.

4.3.1 Approach 1: Regression-based Method

ResNet-50 model architecture is utilized in this approach. The SPEED dataset used in this evaluation has images with pixel size of 1920×1200. In this first approach, direct regression with no prior preprocessing, such as cropping, is adopted. The images are only directly resized to 224×224 to fit the input size of the ResNet model. This makes it more difficult for the network to detect as well as learn the spatial features of the spacecraft within such a small footprint. Nevertheless, the focus of this research is mainly based on the detection-based approaches that will be presented in the next sections. Because there are 11 keypoints, the network output needs an output vector of size 1×22. To achieve this, a global average pooling layer was added after the fifth block, before the output is finally fed into a fully connected (dense) layer with 22 units that gives the final keypoint predictions. Figure 15 depicts this network architecture.

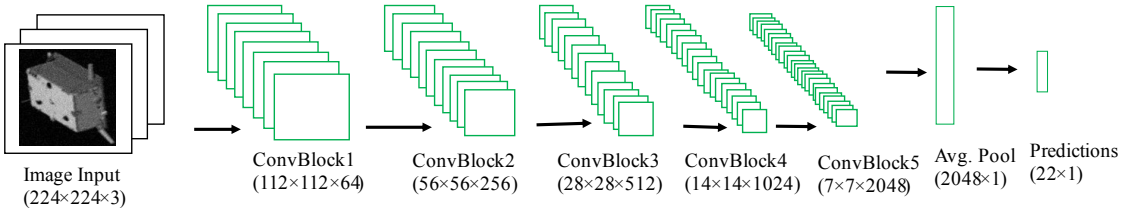


Figure 15: ResNet50 Architecture for Direct Regression.

The network was trained on Google Colab while using an allocated Tesla GPU.

It was trained for 40 epochs in batches of 32. 8000 images were used for training, 1800 for validation, and 200 for testing. As will be discussed in the results chapter, the performance of this network was poor.

4.3.2 Approach 2: Detection-based Method on Full Image

U-Net model architecture is used in this approach. The U-Net is a detection-based network, in that, instead of directly regressing the keypoints, it outputs a heatmap for each of the keypoints. Figure 16 shows this architecture approach. Similar to the ResNet-50 based direct regression, the U-Net network was trained on uncropped images i.e., the full resolution of 1920 × 1200 training dataset images. They were resized to 224 × 224. The network was trained over 40 epochs. The performance of this detection-based approach was much better than the direct regression even though they were both trained on the full image sizes and over the same training epochs. The results for this approach will be presented and discussed in the results chapter as well.

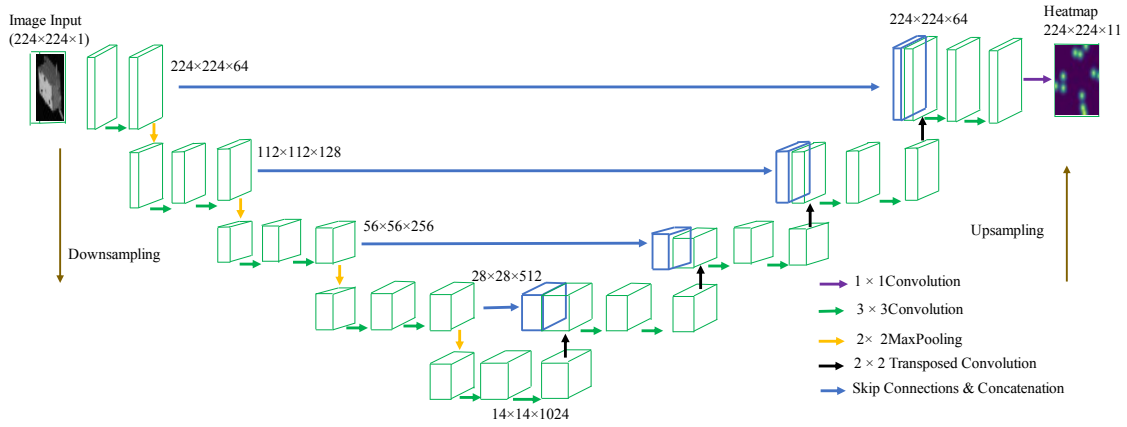


Figure 16: U-Net Architecture for Heatmap-based Approach.

4.3.3 Approach 3: Detection-based Method on Cropped Image

The main focus of this paper is on the inference for the CNN part of CNN-based pose estimation methods, as introduced in Figure 11. To achieve high accuracy in pose estimates, a corresponding high accuracy in landmark detection is required. Hence, in this approach, a framework for achieving high accuracy in keypoint de-

tection is presented. The performance of the keypoint detection is greatly increased by first detecting the spacecraft within the image. The spacecfraft within the image is first detected, as shown in Figure 17. The image is then cropped around the detected box and the keypoint detection network is run on this bounded/cropped image. This approach greatly increases the accuracy of the keypoint detection when compared to the preceding two approaches.

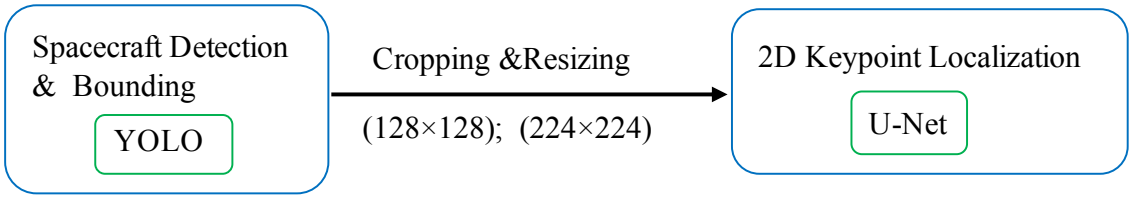


Figure 17: Detection, Cropping and Keypoint Localization Flow.

YOLOv3

The YOLOv3 network is trained for the spacecraft detection phase. The choice of this network is due to its robust and effective performance in many real-time object detection applications [75]. It detects the spacecraft with a single forward pass through the network, hence its fast and efficient. This is very amenable to FPGA implementation and it is supported by Xilinx DPU IP core. The network is trained on a single class, named '*satellite*', since we are only detecting the satellite object within the image. It is trained with the default 9 anchor boxes. The input images are of size 1920 $\times$ 1200. YOLOv3 resizes them to 416 $\times$ 416, detects the object and outputs the bounding box coordinates with the original size as the reference. The images in Figure 18 show performance of the YOLOv3 network in detecting the spacecraft. Though the confidence level is not critical in this application, it shows that the network can detect the spacecraft in varying illumination conditions such as cloudy background.

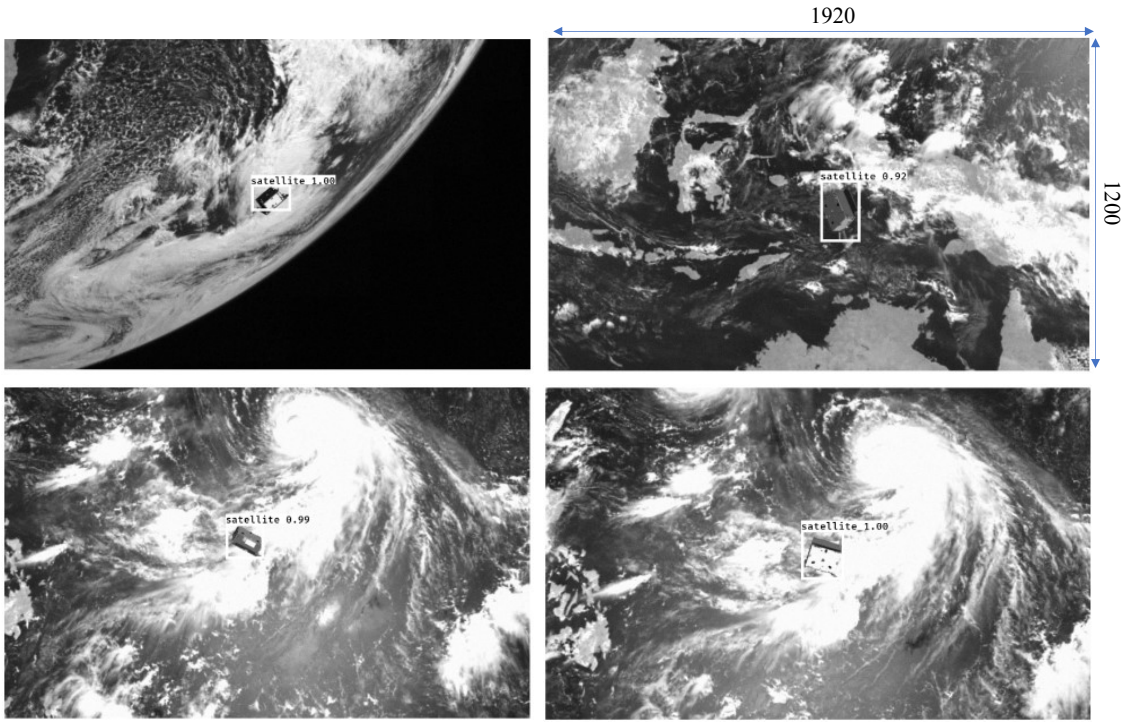


Figure 18: Spacecraft Detection and Bounding with YOLOv3.

Once the spacecraft has been detected and the bounding box coordinates obtained, the image is cropped along the bounding box and fed into the keypoint detection network. This 'zooming in' of the image greatly improves the performance of the keypoint detection algorithms, which, in turn, will improve the pose estimation accuracy. Table 7 shows the ground truth bounding boxes coordinates with the corresponding Yolo-detected bounding boxes. After the spacecraft detection, the images are cropped along the bounding box coordinates and then fed into the keypoint localization network.

Table 7: YOLOv3 Performance in Spacecraft Detection and Bounding.

Images (1920×1200)	Ground Truth Coordinates		YOLO-Detected Coordinates		Normalized Absolute Errors	
	Top-Left	Bottom-Right	Top-Left	Bottom-Right	Top-Left	Bottom-Right
Im7547	817, 476	996, 843	802, 451	1000, 828	0.008, 0.021	0.002, 0.013
Im8575	799, 643	1050, 1064	796, 663	1053, 1053	0.002, 0.017	0.002, 0.009
Im10028	833, 518	1000, 635	899, 512	984, 635	0.034, 0.005	0.008, 0.000
Im10235	809, 497	951, 627	809, 493	948, 623	0.000, 0.003	0.002, 0.003
Im12887	813, 476	1081, 673	798, 483	1082, 665	0.008, 0.006	0.001, 0.007
Im14016	720, 195	1137, 576	712, 198	1153, 585	0.004, 0.003	0.008, 0.008

U-Net

The U-Net model presented in Section 4.3.2 is also utilized in this section. The model utilizes downsampling and upsampling. In down-sampling, the architecture follows a contracting path that captures context. Upsampling is symmetrical to the downsampling path and it is an expanding path that enables precise localization. With the output being heatmaps that rely on pixel localization, the output resolution needs to be high enough to enable accurate localization. Upsampling operators increase the output resolution, leading to greater accuracy in localization. The network is light and fast, making it suitable for onboard FPGA inferencing.

The image input is a cropped image that contains the spacecraft body as detected by the YOLO network in the preceding Section 4.3.3. The image is resized to 128 × 128 and fed into the U-Net model. This input size will determine the output dimensions. A large input size results in a corresponding high output resolution which leads to higher memory requirements during training. Hence, a compromise between the resolution and memory budget is required. The output resolution (and input image size) is chosen to be 128. The network is trained over 40 epochs in batches of 16 on Google Colab.

4.3.3.1 ResNet34 - U-Net Model The performance in keypoint localization can further be improved by a slightly modified decoder-encoder architecture. In this section, a ResNet34 - U-Net model is explored. The encoder (downsample)

part of the network is implemented as ResNet-34. The decoder (upsample) is implemented as the U-Net upsample part. The skip connections from the ResNet-34 are picked at the layers, as shown in Table 8.

Table 8: ResNet34 - U-Net Skip Connections.

Skip Connection	Layer	Dimensions
1	5	$64 \times 64 \times 64$
2	37	$32 \times 32 \times 64$
3	74	$16 \times 16 \times 128$
4	129	$8 \times 8 \times 256$
Base/final	157	$4 \times 4 \times 512$

This model architecture has the highest accuracy performance, hence was picked for the onboard inference implementation. A key aspect in the choice of this network, in addition to its accuracy performance, is that its layers and operations are supported by Xilinx DPU. Figure 19 shows the model architecture.

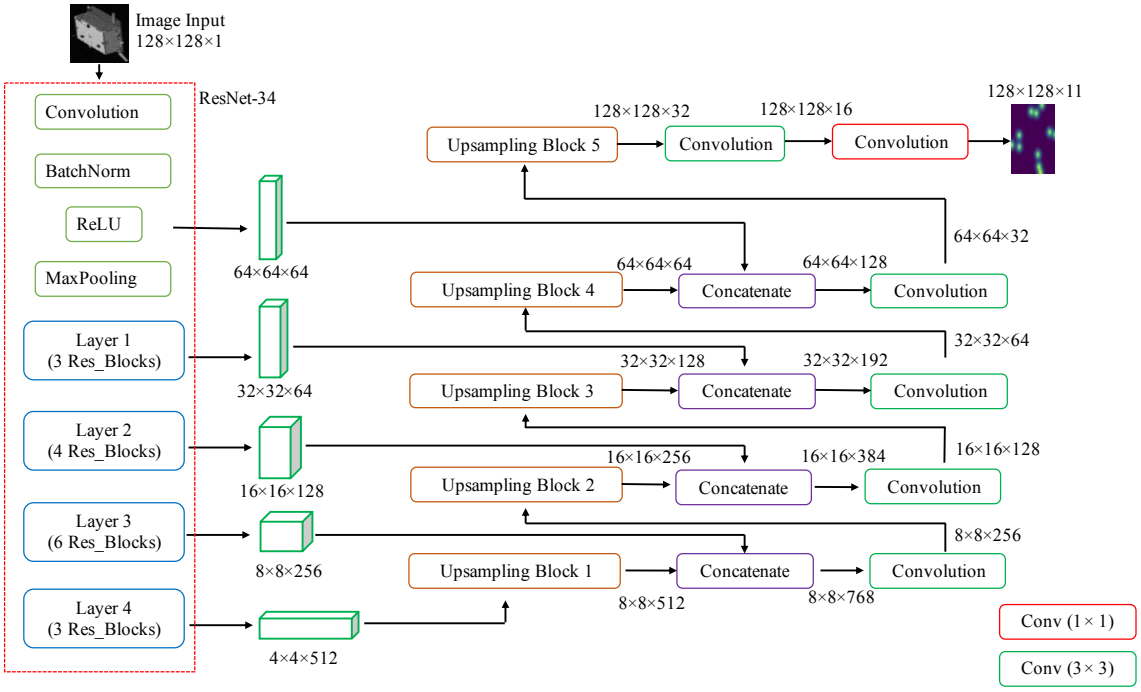


Figure 19: ResNet34 - U-Net Architecture.

5 FPGA Custom-based Inference

This approach is aimed to implement an inference design from the basic FPGA fabric resources. A 3-layer MLP network is implemented for inference. This is due to its relative simplicity compared to a convolutional neural network. The idea was that once an MLP inference design is successfully implemented, it would be easier to implement a CNN design following the same steps. Also, some onboard spacecraft operations can benefit from an MLP network. In this approach, the Kintex- 7 KC705 Evaluation board in Figure 2 is used as the target platform.

Methodology Goal

This design approach aims at a modularized and parameterized implementation of MLP networks on FPGA. Through this approach, networks of different sizes, types, activation functions and other constraints can easily be implemented on FPGA without consuming much time on HDL coding. This implementation also decouples the underlying FPGA architecture by not directly using any of the vendors' specific IP cores. This allows for a cross-FPGA platform implementation by using inferencing that leaves it to the synthesizing tool of each vendor to decide the appropriate IP cores for the network. This design framework is tested on two different applications: a simple data classification and MNIST dataset.

The objective of this framework is design of a modularized network that allows easy modifications without time-consuming and complex changes in the HDL code. This modularized design also allows for flexible activation function implementations, hence various activation function implementations and combinations between layers can be easily explored. This is because the activation function is decoupled from the layers as a standalone module. The various components of the network i.e. layers, neurons and activation function are implemented as standalone modules that are then linked together by a control module.

5.1 Training and Inference Simulation in MATLAB and Simulink

The data used for this 3-layer MLP was based on MNIST dataset. MATLAB was used for the training of the network. This step gives the weights to be used in the inference phase. The MATLAB scripts used for the training phase have included in the appendix of this thesis. Once the network has been trained and weights obtained, the network can be processed for inference. For hardware inference, there are several considerations to be made. The key factors that need to be addressed are:

- Weights storage
- Fixed-point vs floating-point data representation
- Efficient hardware implementation of the activation function (sigmoid)

To aid in addressing the above issues, Simulink was used. After the training and generation of weights in MATLAB, they were converted from floating point to fixed point. The network was then implemented in Simulink. The activation function was also implemented at the gate level as would be in the FPGA itself. Hence Simulink was used to model, simulate and evaluate the fixed-point network before porting to the FPGA. This modeling gives a visual implementation of the network to help in the HDL implementation at the hardware implementation level. Figure 20 and Figure 21 shows the floating point and fixed point simulation, respectively, in simulink.

Floating Point Network Simulation on Simulink

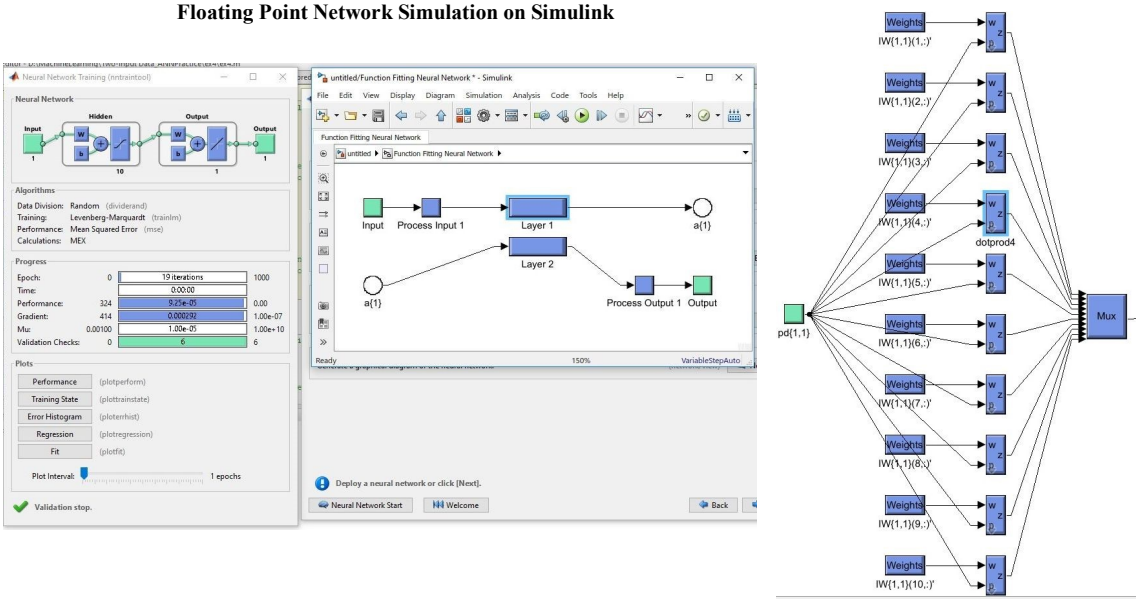


Figure 20: Floating Point Simulation in Simulink

Fixed Point Network Simulation on Simulink

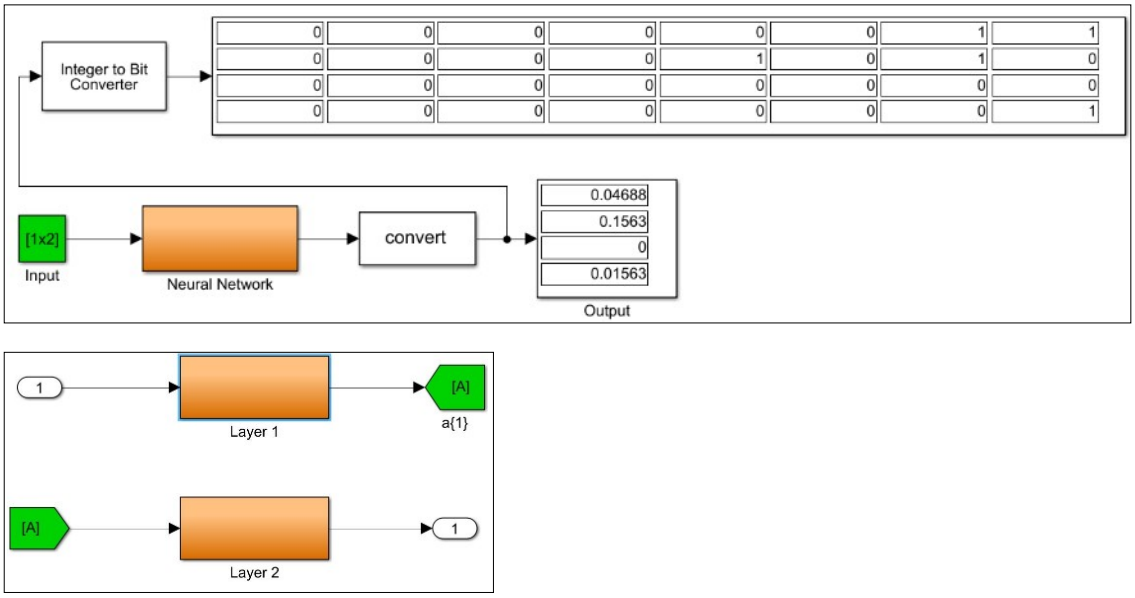


Figure 21: Fixed Point Simulation in Simulink

5.2 Preparation for Hardware Inference

5.2.1 Activation Function Hardware Implementation

The sigmoid function presented in Equation (5) was used as the activation function in the MLP network. The activation function at inference needs to be same as the one used in the feedforward implementation. However, implementing the function directly on the FPGA is area and resource expensive. This is due to the division and exponentiation requirements. Two of the main methodologies of implementing such functions are:

- Piece-wise Linear Approximation
- Look-up Table (LUT)

The LUT approach is memory intensive as the LUTs form the basic memory logic in the FPGA fabric. Hence the piecewise linear approximation was used. In [76], it is implemented as a piecewise linear approximation of a nonlinear function (PLAN). This approach directly maps the input,x, to the output, y using digital gates. The PLAN sigmoid approximation compared to exact sigmoid implementation is shown in Figure 22.

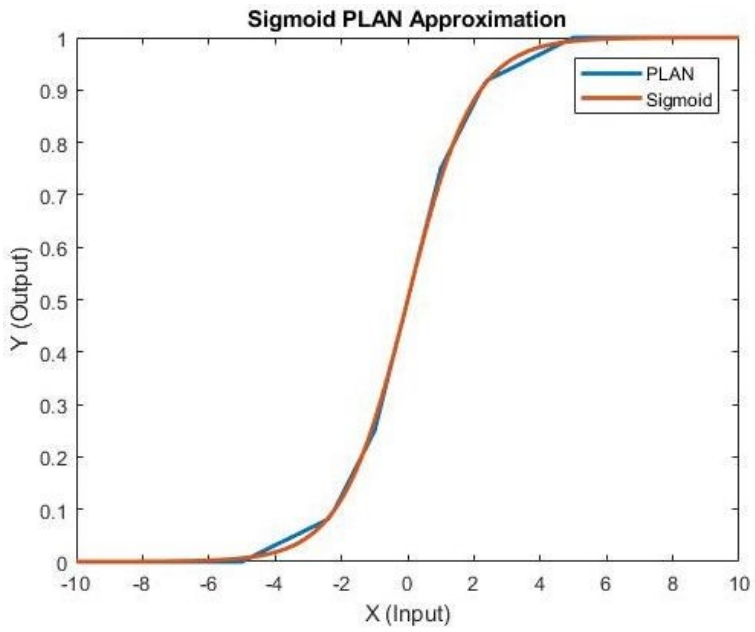


Figure 22: PLAN vs Floating-point Sigmoid Implementation

A similar approach has been used in [77] where it is implemented using digital gates for an entirely combinational approximation, named SIG-sigmoid. Both of these implementations have been tested in this paper since the network design is highly modularized for easy adoption of hardware implementation of different activation functions and/or different hardware implementations of a particular activation function. The PLAN architecture is shown in Figure 23

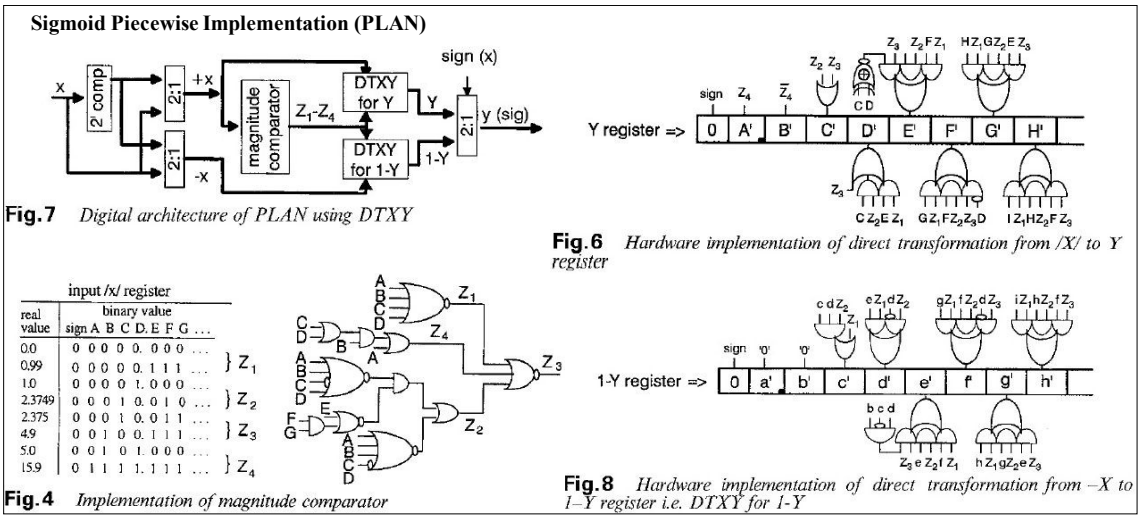


Fig.7 *Digital architecture of PLAN using DTXY*

Fig.6 *Hardware implementation of direct transformation from /X/ to Y register*

Fig.4 *Implementation of magnitude comparator*

Fig.8 *Hardware implementation of direct transformation from –X to 1–Y register i.e. DTXY for 1-Y*

Figure 23: PLAN Sigmoid Architecture

With the PLAN implementation, the sigmoid curve is linearly approximated using 5 straight lines. It eliminates the exponential term and offers a gate-level implementation. This implementation is simullated in Simulation as shown in Figure 24.

5.2.2 Fixed Point Representation

As presented earlier, inference on FPGA is computationally demanding. To reduce the amount of FPGA resources and power consumption, fixed-point data representations are preferred to floating points. More so, due to data normalization and limited dynamic range of the weights, fixed points are adequate for acceptable accuracy of most machine learning algorithms. The common fixed-point notation used is $Q_{m.n}$; where m is bits for the integer portion and n is bits for the fractional part. This notation requires that the number is 2's complement i.e. signed.

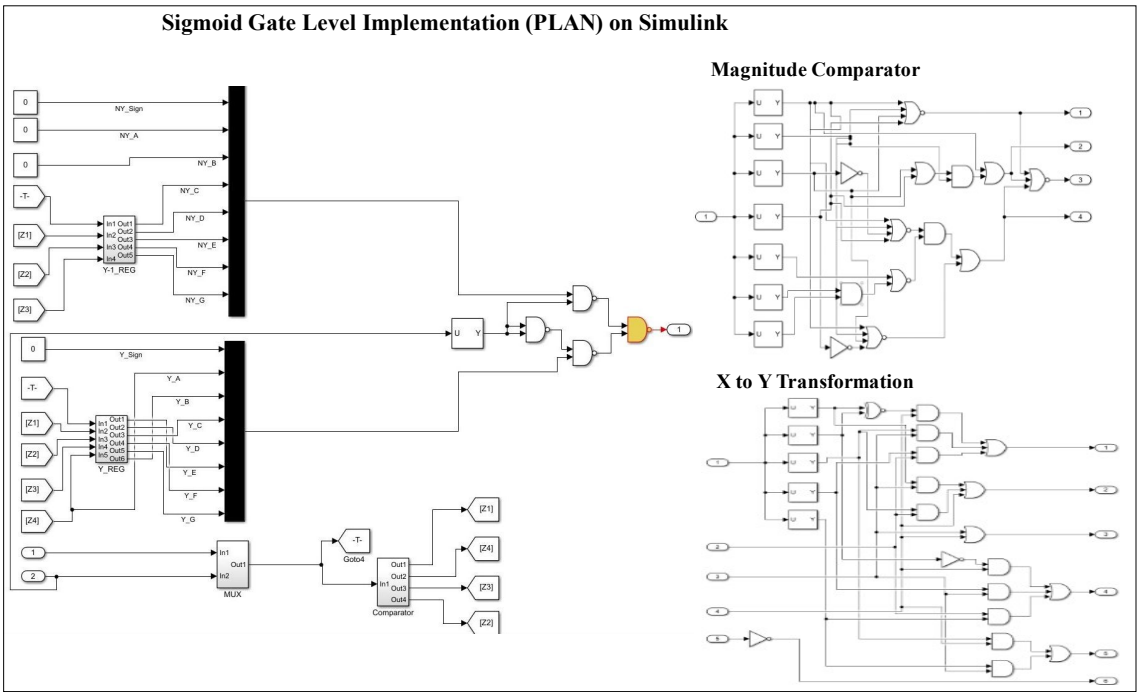


Figure 24: PLAN Sigmoid Implementation in Simulink

Fixed-point representation is of the form shown in Figure 25

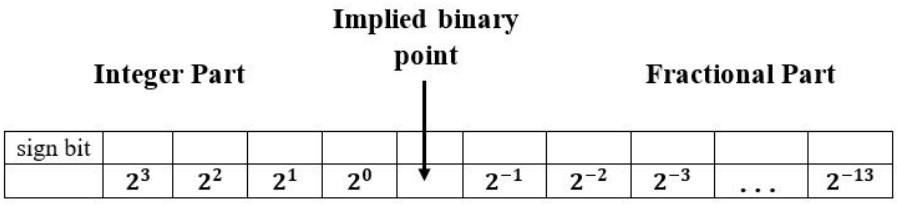


Figure 25: Fixed-point Representation

Using the above notation, total number of bits required to represent signed fixed point number is $N = m+n+1$. Examples of such representations are given below:

- $Q_{3.6}$ - This is a 10-bit number; 3 bits are used for integer part, 6 bits are used for fractional part and 1 bit used for sign representation.
- $Q_{7.0}$ - This is an 8-bit Integer number; 7 bits are used for integer part, 0 bits are used for fractional part and 1 bit used for sign representation.
- $Q_{0.15}$ - This is a 16-bit Fractional number; 0 bits are used for integer part, 15 bits are used for fractional part and 1 bit used for sign representation.

Key Concepts of Fixed Point Representation

Dynamic Range: This is the ratio between the largest number and the smallest positive number that can be represented by the fixed point number. It is only dependent on the word length i.e. the total number of bits in the number. Hence for a $Q_{m.n}$ number, the dynamic range is given in Equation 13.

$$DynamicRange = \frac{MaxNum}{MinNum} = 2^{N-1} - 1 \tag{13}$$

Precision: Maximum number of non-zero bits that can be represented by a fixed point number. It is equal to the word length i.e. total bits in the number.

Resolution: This is the smallest non-zero magnitude that can be represented by the fixed point number. It is dependent on the fractional part such that: $Resolution = 2^{-n}$ where n is number of bits representing fractional part.

Accuracy: This is a factor of the resolution and signifies the maximum difference between a real value and its representation using the fixed point. It is given as: $Accuracy = \frac{Resolution}{2}$

Arithmetic Operations on Fixed-Point

Multiplication: In multiplication, the result is $N_a + N_b$ bits where N_a and N_b are the number of bits in the two numbers being multiplied. However, all these bits in the result may not be desirable for further use in the machine learning algorithm. Hence the result is often resized and the number of bits reduced.

Addition: Unlike in multiplication, the result in addition (substraction) requires one additional bit compared to the largest input number. When two or more numbers are added (subtracted), the total number of bits required in the final result is given by $N = m + log_2(x)$ where m is the width of the numbers being added (assumed that the numbers are of the same width) and x is the number of elements being summed.

5.3 Hardware Inference Design

In Kintex-7 XC7K325T FPGA, the multiply and accumulate (MACC) core computation of a neural network is efficiently performed in the DSP48E1 slice. This is a 25 x 18 two's complement multiplier with a 48 bit accumulator. This means that the inputs and weights which are the inputs to MACC block should have maximum data width of 25 x 18 bits interchangeably. Usage of shorter data width will be beneficial in other ways but will not reduce the number of DSP48E1 blocks to be used. The fixed point data width is parameterized consistently across all the network modules to allow easy modification in data precision. However, low data width may adversely compromise the network accuracy. It is shown statistically in [78] that a feedforward operation requires a minimum precision of 8 bits.

The neural network is trained off-line on MATLAB. The weights obtained are in double precision and are first converted to fixed point representation as presented in the preceding section. The number of bits chosen will have an effect of the performance of the network once synthesized on the FPGA. In this implementation, the weight width chosen is 18 bits with 1 sign bit, 4 integer bits and 13 fractional bits, that is $Q4.13$.

The trained network's weights are loaded into the FPGA BRAM; which in Xilinx 7 series FPGAs stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs or one 36 Kb RAM. Therefore using 18 bits for the fixed point weights, the network can be designed for each BRAM to store a maximum of 1K or 2K weights. In the parameterized network architecture, each layer has its associated BRAM. This is quite effective for a MLP network that will rarely need more than a thousand weights per layer. This implementation prioritizes node parallelism which is achieved by all neurons in the same layer processing their outputs simultaneously [76].

This, however, can be limited by the number of neurons in the layer and target FPGA resources. Nevertheless, a 3 layer MLP with less than 400 neurons per layer (hidden and output layers) can be fully implemented by this architecture for Xilinx 7 series FPGAs. Reading from the BRAM requires a clock edge and only a single address (weight) can be read at a time for a simple dual port and two weights for a

true dual port. Thus in order to achieve node parallelism where all the neurons in the same layer access their weights simultaneously, a cache-kind of memory needs to be implemented. Hence, each neuron has its weights cache that is implemented as a FIFO register using distributed RAM instead of the dedicated BRAM. On power up, the weights for each layer are stored in the FIFO to speed up the access by the neurons during operation. On initialization, each neuron stores its relevant weights. This is useful in that the neurons can process the MACC operations simultaneously and provide their outputs concurrently during same clock cycle.

All the control is centralized at the layer level to save on resource utilization and to make it independent of the size of the network. Hence fetching of data by each neuron and its subsequent MACC operations are controlled by a single control module for each layer. Addition of more neurons does not require any additional control logic. The neurons have been designed to utilize a single DSP48E1 block per neuron and a single FIFO register as a cache for the stored weights. Thus every addition of a neuron requires one more DSP block and FIFO register which is implemented using look up tables (LUTs) of the FPGA fabric. The number of LUTs required is dependent on the number of weights for a particular neuron. For example, a 3 layer digit classification network with 400 inputs,25 neurons in the hidden layer and 10 output neurons requires 35 DSP48E1 blocks. The controller is implemented as a Finite State Machine (FSM). The 3-layer MLP network and the FSM controller are shown in Fig. 26.

Each layer has a FSM controller which eases modularization. FSM_{n+1} is activated when the previous layer (n) completes its computations and its outputs are ready to be fed into the next layer. FSM_{n+1} activates $layer_{n+1}$ and controls the operations in this layer till all the MACC operations have been completed. It then sends a signal to $FSMn+2$ to start $layer_{n+2}$ operations. The FIFO of $layer_{n+2}$ loads the neuron outputs of $layer_{n+1}$. For the input layer, $Neuron\_En$ enables the neuron operation after BRAM has finished loading the weights into cache after power up.

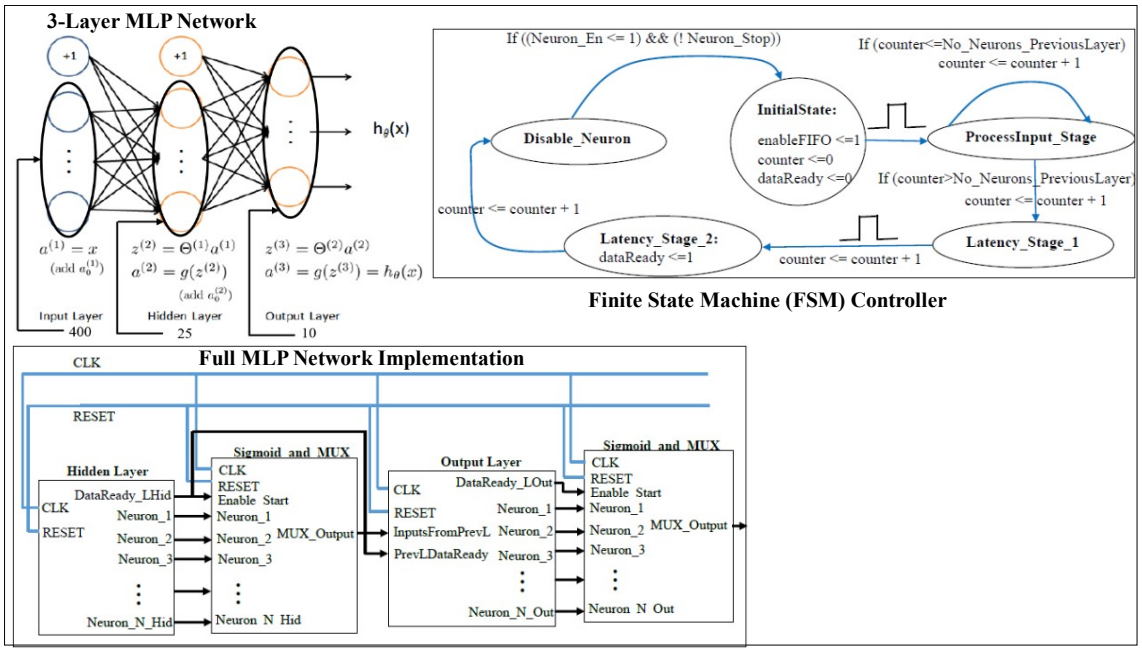


Figure 26: MLP Network Implementation and FSM Controller

For hidden layers, this is after the previous layer has signaled that it has finished its operation and its neurons' outputs are available to be fed into the next layer. *Neuron_Stop* signifies that the neurons in current layer have finished their operations. This stops any further loading of weights and inputs and also stops any further calculations in the DSP MACC.

Results of the calculation are maintained as the layers' outputs until a new round of calculation begins. The latency stage is added to enable the neuron to hold the correct result at its output for the next layer to read before it transitions back to the initial stage. This output is available to the next layer after two clock cycles hence the two latency stages. This latency is due to the MACC operation in the DSP block in which the correct result is available after the third clock cycle once the last inputs have been fed into the block.

The full network architecture is shown in Fig. 26. It is made up of three distinct modules: $hiddenlayer, outputlayerandsigmoidandmultiplexermodule$. These are shown in Fig. 27.

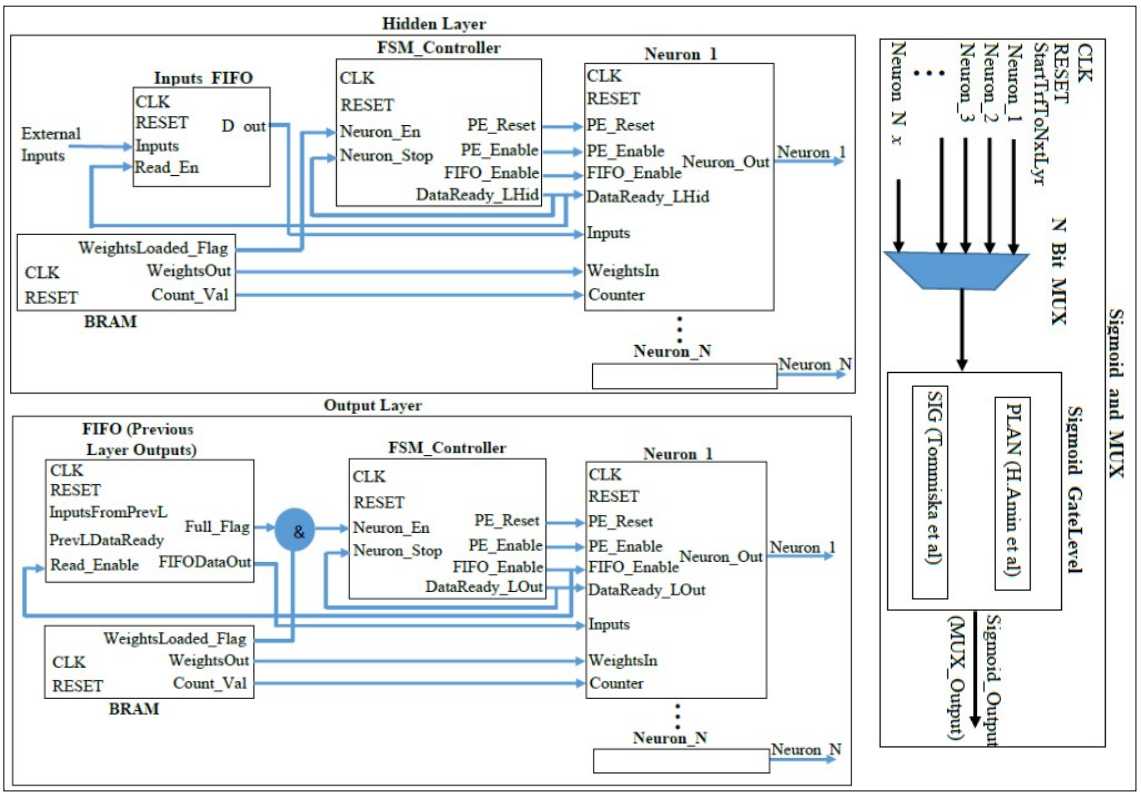


Figure 27: Hidden & Output Layers and Sigmoid Module

Use of additional layers would only require replication of the hidden layer and sigmoid modules. On power up, the weights are loaded from the BRAM into FIFOs of the respective layers. However, only the hidden layer module is activated at this point. Once all the neurons in the hidden layer have completed their operations, the flag $DataReady\_LHid$ is set to high; consequently enabling the sigmoid and output layer simultaneously through $Enable\_Start$ and $PrevLDataReady$ flags respectively. The output layer thus begins obtaining data from the sigmoid and multiplexer block, and concurrently carrying out the MACC operations. Once all the calculations are finished, the output layer will set its $DataReady\_LOut$ flag high, which enables the sigmoid module and the final ANN prediction results are now available on the MUX_Output port for external application and use.

6 FPGA IP-based Inference

The direct implementation approach presented in the custom-based flow is not effective for deep learning architectures with many layers. The custom implementation is time-consuming and prone to failure due to tight timing constraints in FPGA fabric. Additionally, such an implementation is not scalable to different network architectures since its highly customized. Hence a more optimized hardware implementation was pursued based on Xilinx Deep Learning Processor Unit (DPU), which is a recent IP introduced by Xilinx in 2018.

Methodology Goal

In this approach, the goal is to implement inference of the CNN part of the CNN-based Pose Estimation algorithm developed in 4. The target device for this implementation is the Xilinx Zynq UltraScale+ MPSoC. This family of devices incorporates both programmable logic (PL) and processing system (PS) on a single chip. MPSoC implies presence of multiple (and heterogeneous) processors. These chips include a real-time processing unit (RPU), which is a dual-core Arm Cortex-R5F, and an application processing unit (APU), which is a dual- or quad-core Arm Cortex-A53. The 64-bit APU enables high-level operating system support, e.g., Linux. The Ultra96v2 board 3 is used as the evaluation board. It is a low cost board that incorporates the Zynq MPSoC and other essential peripherals. In a real space mission, the MPSoC device can be incorporated into a custom PCB design that only utilizes the necessary peripherals. This would result in an optimal implementation that meets the key aspect of this work: onboard inference hardware that is small, low-power, and low-cost for satellite operations.

The flow for this methodology is depicted in Fig. 28.

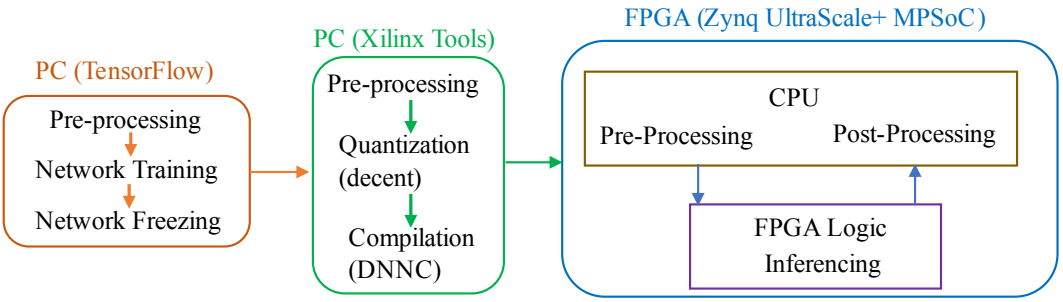


Figure 28: IP-based Inference Flow

6.1 Xilinx DPU IP Core

The DPU is a Xilinx IP core for implementing FPGA-based inference accelerators of deep learning architectures. It is a configurable computation engine that is optimized for convolutional neural networks. It is intended for implementation in the programmable logic of Xilinx Zynq-7000 SoC and UltraScale+ MPSoC devices. It requires devices with direct connections to a processing system. For its operation, it requires an application processing unit that services interrupts from the DPU core and coordinates data transfers between the DPU core and other peripherals. It also needs access to memory locations, such as RAM for input images, temporary, and output data. The DPU core is made of processing elements (PE) that consist of FPGA building blocks, such as multipliers, adders, and accumulators.

DPU IP can be configured for the optimized implementation of various CNN architectures. Some of the user-configurable parameters are briefly presented here. The first parameter is the number of DPU cores that can be instantiated in the FPGA fabric. A maximum of three cores can be instantiated in a single DPU IP. An increase in the number of cores results in increased consumption of PL resources and power requirements. Another parameter for DPU customization is the convolution architecture. The Xilinx IP offers eight architectures to choose from. These are B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096. These architectures are based on the three dimensions of parallelism in the DPU convolution architecture: pixel, input channel, and output channel parallelisms. The naming convention comes from the peak operations per clock for each of the architecture configurations. For example, B512 offers 4, 8 and 8 pipelines for pixel,

input channel and output channel parallelism respectively and this results in 512 peak operations per clock.

The DPU architecture and its configuration menu in Vivado is shown in Fig. 29.

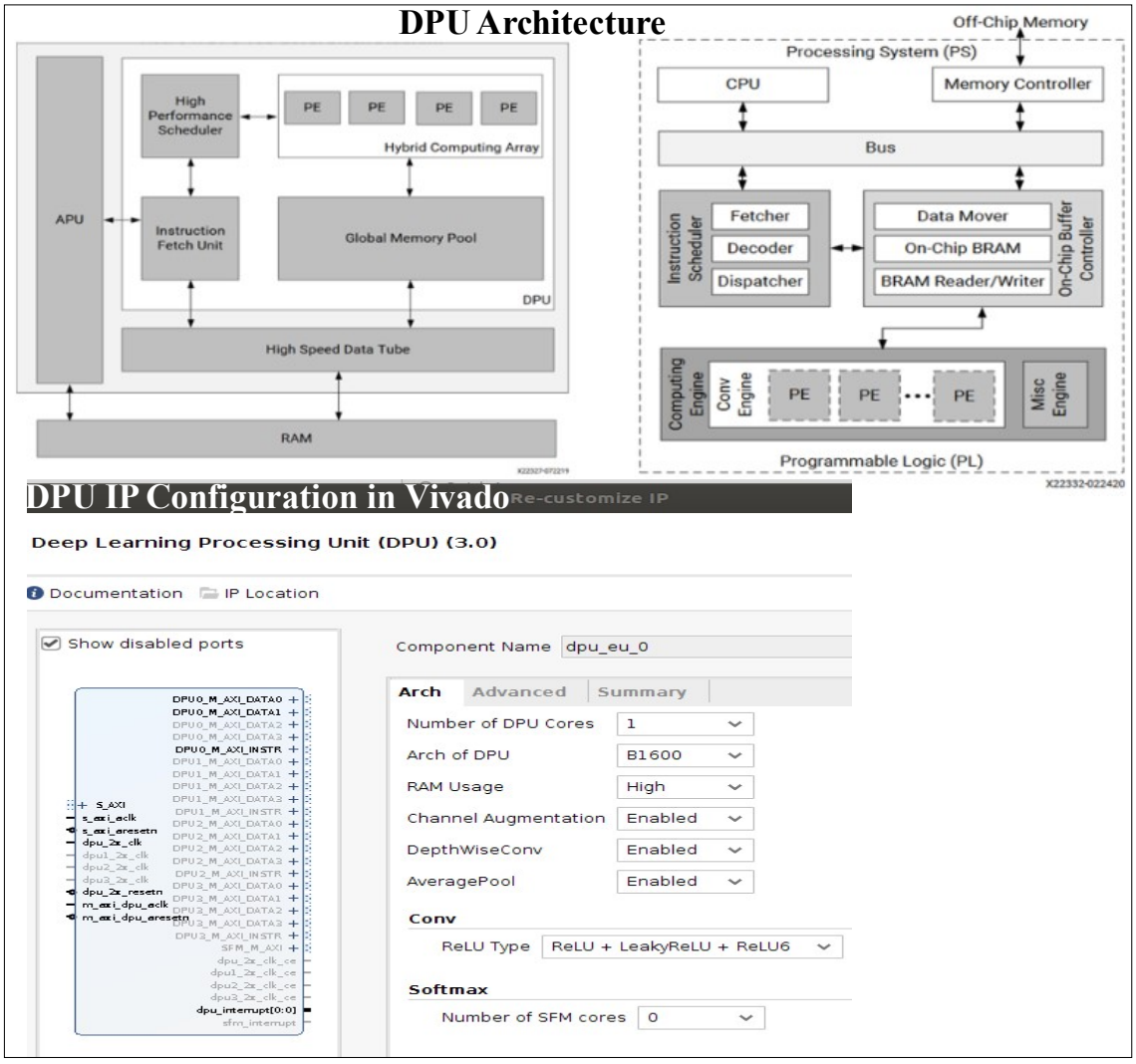


Figure 29: DPU Architecture

RAM usage is another parameter that can be configured to set the total amount of on-chip memory that can be used in different DPU architectures. This on-chip memory is for buffering weights, bias, and intermediate network features. The DPU IP also allows enabling or disabling of Softmax instantiation. Enabling Softmax will allow implementation in hardware resulting in fast inference but greater

consumption of PL resources. Other parameters available for configuration include ReLU type, AveragePool, Channel Augmentation and Depthwise Convolution.

6.2 Zynq MPSoC Hardware and Software Implementation

Hardware design is implemented in Xilinx Vivado IDE. The DPU IP is instantiated in appropriate configurations in order to meet the ResNet34 - U-Net network architecture that was presented in 4. In addition to the DPU IP, other IPs, such as Zynq MPSoC, Processor System, Clock Wizard, and others, are also instantiated to meet the hardware requirements for a fully operational device. Vivado enables the instantiation of IPs in the PL part of the device. The PS part is accessed from the software development side/phase. Figure 30 shows the block design in Vivado.

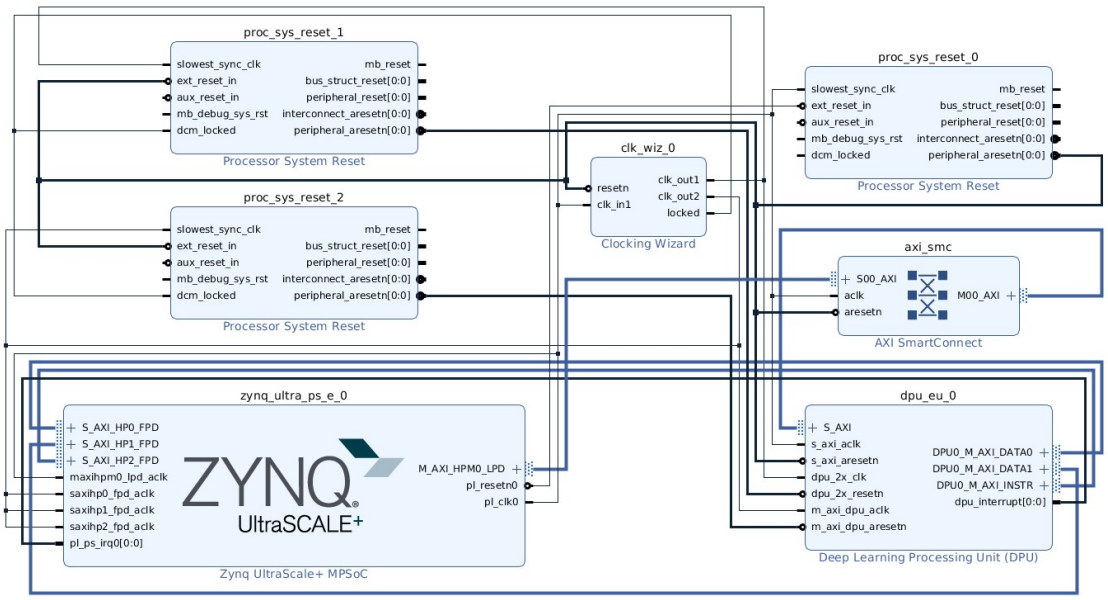


Figure 30: Hardware Block Design in Vivado.

The hardware design is synthesized and implemented. A bitstream file is generated and the hardware design is exported to be used as the base hardware upon which the software is built. Data between the PS and PL regions is exchanged via the PS-PL interface, which is AXI4-based. In a real system, the input data would be obtained from a camera. However, in this implementation, the images are stored in an SD card, from which the PS accesses via the linux file system and then inputs

them to the DPU during inference. The PS has other input/output peripherals, such as USB, UART, DisplayPort, etc., which enable data input and visualization on external displays. These interfaces are also exposed by the Ultra96v2 board.

Yocto/Petalinux is used to generate a custom Linux image that incorporates the deep learning aspect of the project. Xilinx DNNDK tool, which is required to deploy neural networks on the DPU, is incorporated as a package in the Yocto flow. DNNDK is a full stack deep learning tool-chain for inference with the DPU and it is composed of the following components: Deep Compression Tool (DECENT), Deep Neural Network Compiler (DNNC), Neural Network Runtime (N2 Cube) and DPU Profiler. The DPU instructions which are closely tied to the DPU architecture, target Neural Network and the AXI data width are generated offline using DNNC. This results in an Executable Linker File (ELF), which is then compiled together with other custom C/C++ program instructions that control other tasks of the deep learning algorithm, such as loading of images, visualization, and other pre-processing tasks. The CPU and DPU elf files are then compiled into a single file that is loaded into the MPSoC. Figure 31 shows the flow for hardware, Linux image generation and the flow for DPU-based deep learning implementation.

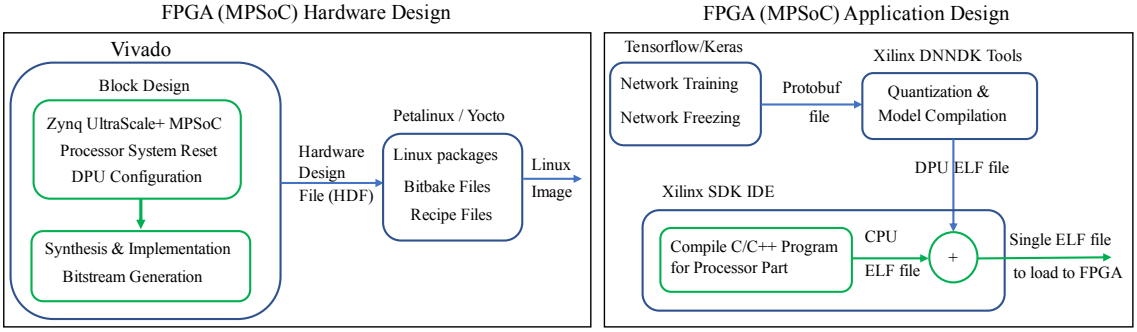


Figure 31: Hardware and Software Implementation Flow for MPSoC Inference.

6.2.1 ResNet34 - U-Net Architecture Inference Implementation

Network Freezing

The first step in preparing the trained network for inference is to freeze it after training. During training, a model stores not only the trainable weights, but also metadata essential in the training phase. However, for inference, this metadata

is not required. Hence, freezing the network achieves two main objectives: the weights are 'frozen' i.e., made untrainable and metadata not required for inference is removed. This results in a light-weight model graph that is efficient for inference.

Network Quantization

The frozen weights and graph variables are in a floating point. Many deep learning models have tens of millions of weights and, thus, floating-point representation results in huge data size. For inferencing, especially on edge devices such as FPGAs, there is limited memory for weights and data storage. The weights can be converted to lower precision and fixed-point representation such as 4-bit, 8-bit, or 16-bit integer representation. This greatly reduces the size of the data and overall model size, with little degradation in the model accuracy. Xilinx utilizes INT8 and INT4 optimization in its DSP48E2 slices for deep learning operations in the DPU. Quantization improves on memory bandwidth and power efficiency.

Xilinx provides the *decent_q* tool for quantization purposes. It takes the frozen graph as input. The key parameters for *decent_q* are the input and output nodes, input shape and the pre-processing input function. It also requires a few images for calibrating the quantized network. A deploy model is generated after successful calibration. In the deploy model, the upsample layers are implemented by Xilinx's custom *DeephiResize* layers. The rest of the layers, have similar implementation to the frozen Tensorflow models.This is ready for the next phase of compiling i.e., generating the executable linker files.

Network Compilation

After quantization, the network is compiled for deployment to the hardware (DPU). The Deep Neural Network Compiler (DNNC) is used in order to generate ELF files and kernel information for deployment. The information gives a summary of the layers and operations that have been compiled for DPU and the operations that need to be deployed on the CPU side. Operations, such as softmax, sigmoid, and average pooling, are best suited for CPU deployment; hence, DNNC does not compile them into the ELF files. Figure 32 shows the deploy model that is generated in the quantization stage and the DNNC output of the compilation stage.

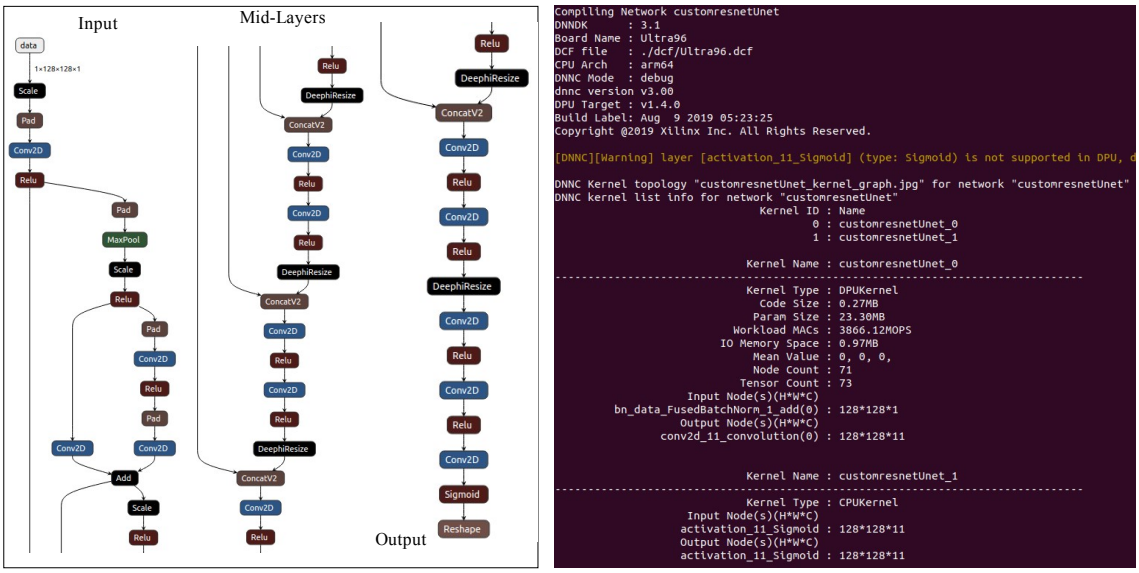


Figure 32: Deployment Model (from quantization stage) and Compiled Network Summary by DNNC.

Network Deployment

The hardware and software flow that is presented in Section 6.2 and Figure 31 is followed for the deployment to the MPSoC device. There are mainly two approaches for deployment to the FPGA board. One is to use the DNNDK tools directly in the Xilinx SDK by incorporating the DPU ELF file into a C/C++ application. The other is to use the Vitis-AI flow that greatly eases the deployment process. The latter has support for both Python scripts and C++ applications for the CPU side. The use of Petalinux on the CPU side also enhances the integration between the CPU and DPU architectures. Vitis-AI provides Python and C++ APIs, at both the high-level and low-level. The latter enables closer manipulation of the DPU engine. This includes DPU kernel and task creation and destruction; the manipulation of DPU input and output tensors; and, the deployment of DPU un-supported operations to the CPU side. These APIs also enable implementation of pre-processing of input data and post-processing output data for the DPU. The sigmoid part of the network is implemented on the CPU side in a Python script. The output has 11 channels representing the heatmaps for each of the 11 keypoints. This is reshaped on the CPU side into a single channel i.e., single heatmap.

7 Results

7.1 Results of Keypoints Localization in CNN-based Spacecraft Pose Estimation

7.1.1 Approach 1: Regression-based Method

As presented in 4.3.1, the ResNet50 network was used for direct regression on the full image resolution. The original image sizes of 1920 × 1200 were directly resized to 224 × 224. Figure 33 shows the performance of the network after 40 epochs. The network has to localize the spacecraft within the image and learn the spatial coordinates of the spacecraft keypoints. As expected, the performance is poor due to the small size of the spacecraft in comparison to image size. As earlier noted, the performance could be improved by more training and use of more data. Nevertheless, from literature review, the performance of regression-based approaches would still be poor. Hence, this approach was not developed further.

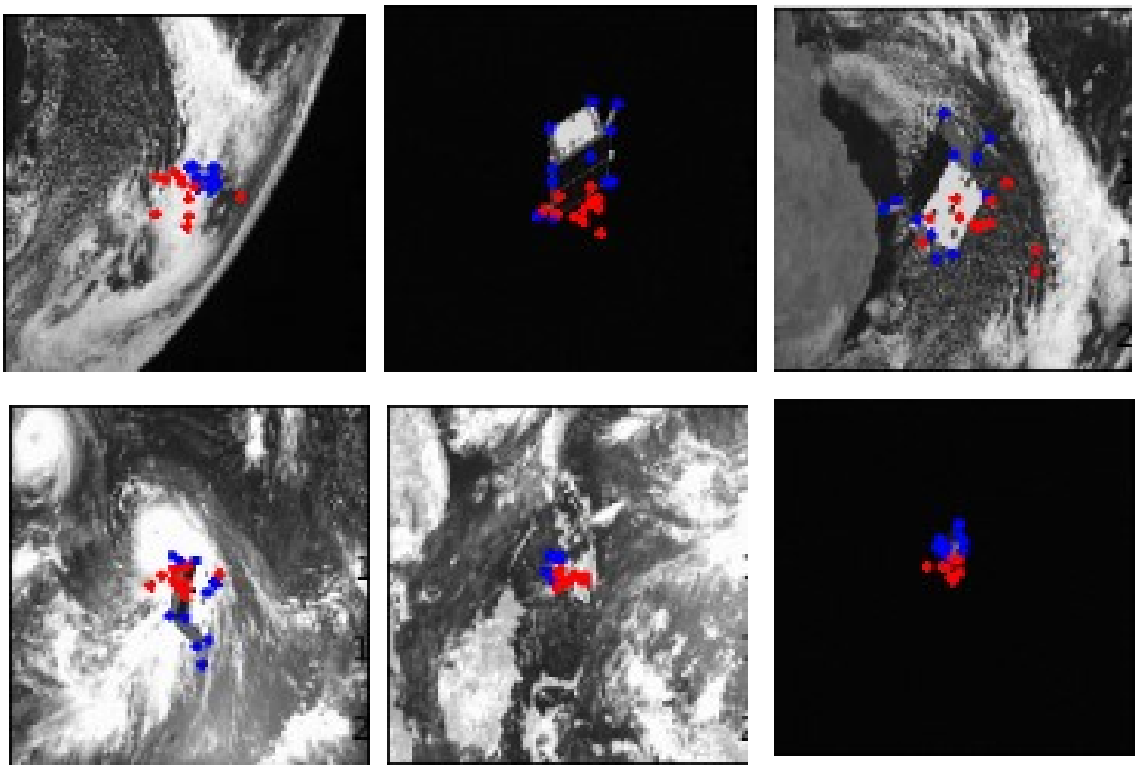


Figure 33: Direct Regression Performance after 40 Epochs Training.

7.1.2 Approach 2: Detection-based Method on Full Image

U-Net architecture is used in this detection-based approach. The output of the network model is a heatmap from which the coordinates of the keypoint are obtained. Similar to approach 1, the original images were resized to 224 × 224. Figure 34 shows the U-Net model performance on the uncropped images. Though the performance is still poor, it is significantly better than the ResNet50 based direct regression. Its keypoint detection RMS Error is averagely two times less than the one based on ResNet50.

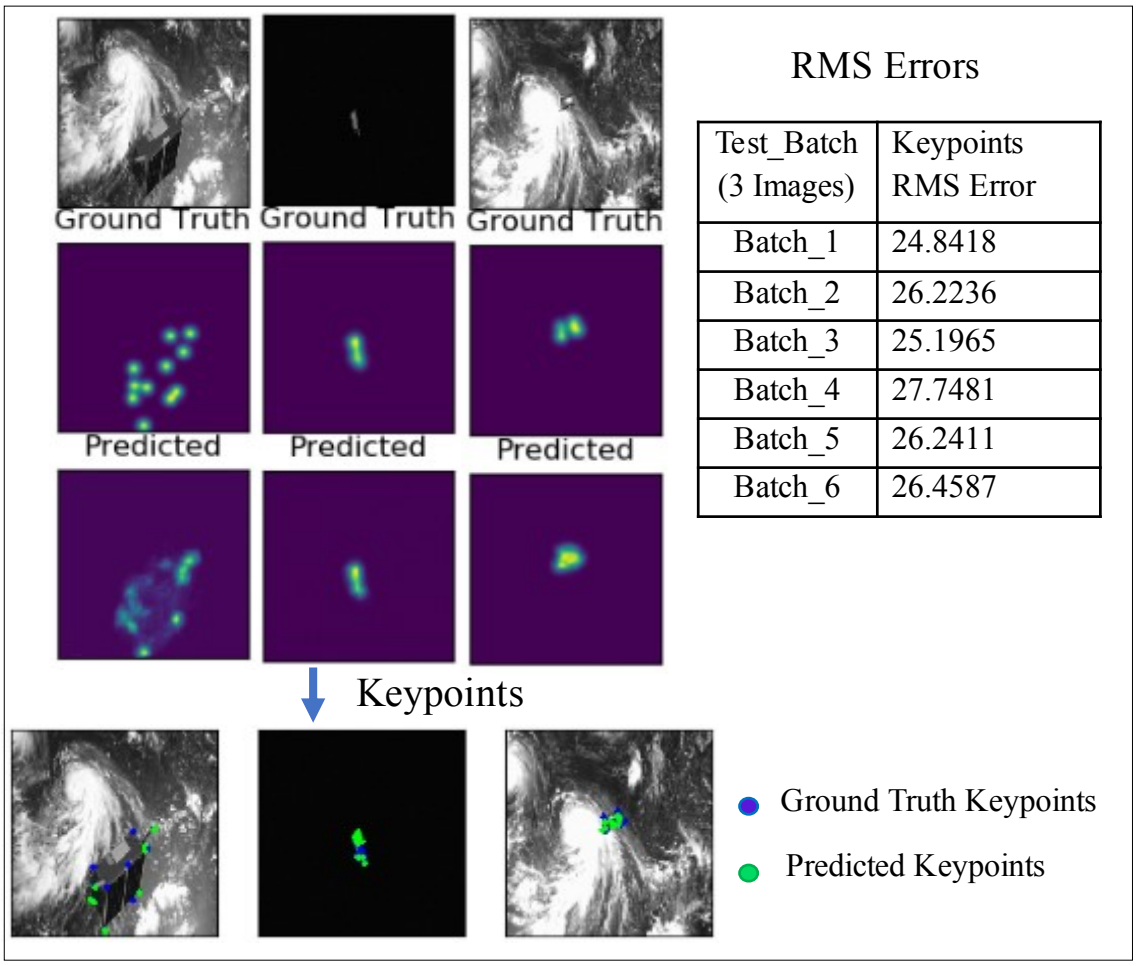


Figure 34: U-Net Model Keypoint Detection Performance on Full Images.

7.1.3 Approach 3: Detection-based on Cropped Image

7.1.3.1 U-Net Results

From the previous approaches, it is noted that the small size of the spacecraft within the image makes it difficult to predict the keypoints. In this 3rd approach, the images are first cropped by use of the YOLOv3 algorithm to detect the spacecraft within the image. After cropping, the images are resized to 128 × 128 and fed into the U-Net model. The network is trained over 40 epochs in batches of 16 on Google Colab. Images in Figure 35 show the performance of the U-Net model in keypoint localization on cropped images. This performance on cropped images is greatly superior to the same model's performance on uncropped images, as previously presented in Figure 34.

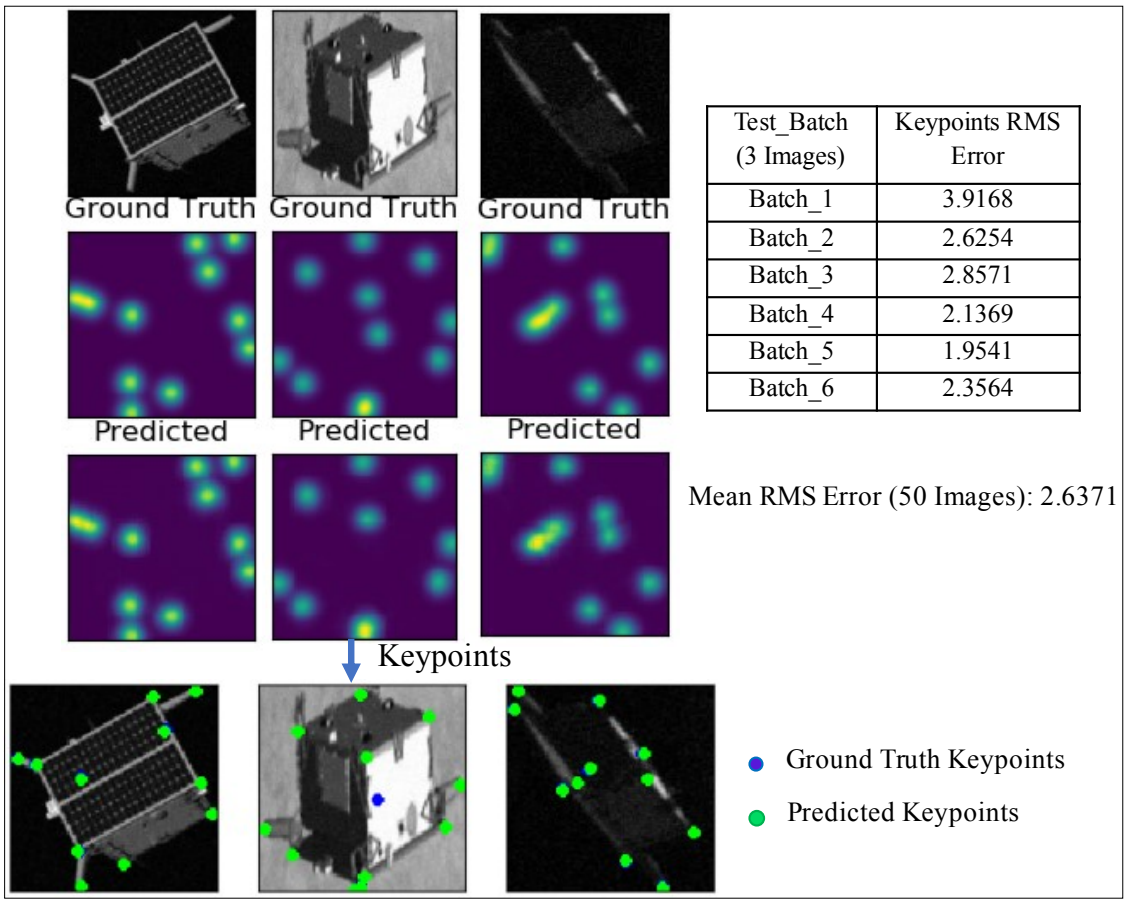
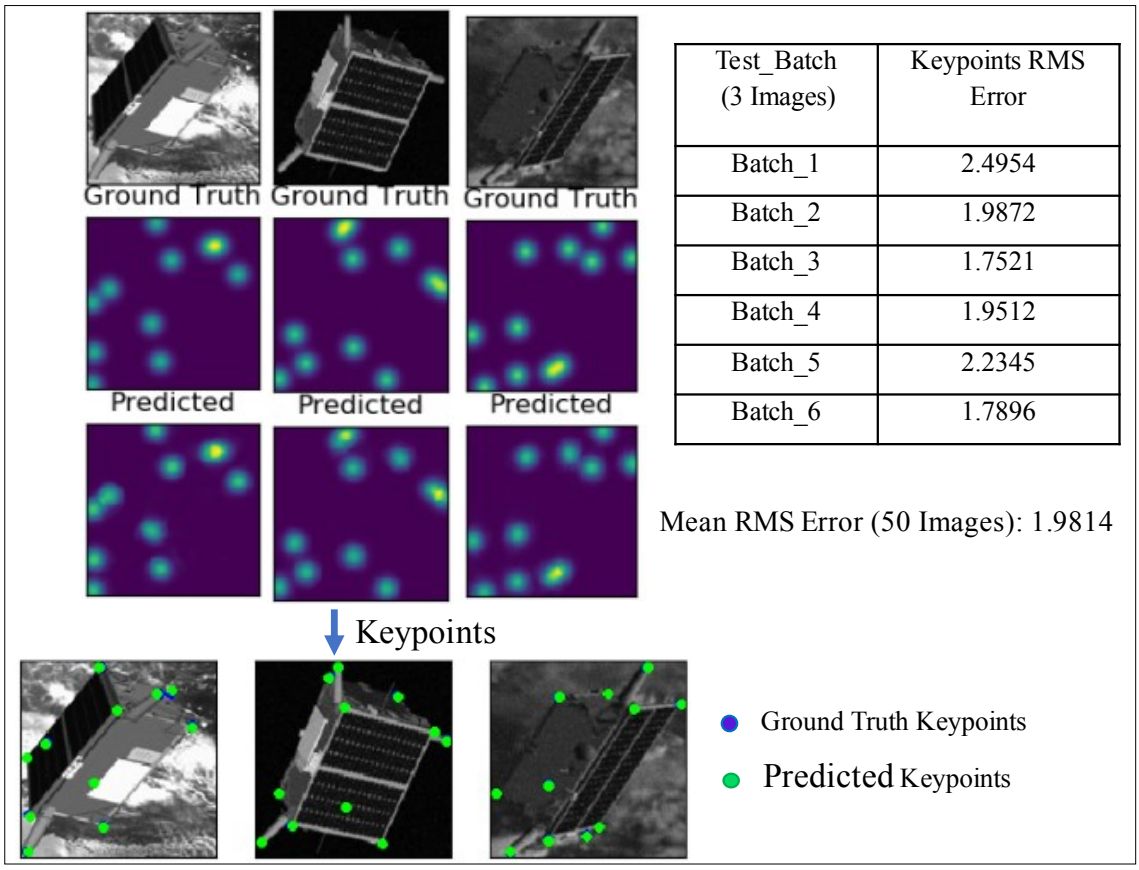


Figure 35: U-Net Model Keypoint Detection Performance on Cropped Images.

7.1.3.2 ResNet34 - U-Net Results

This decoder-encoder model has a relatively better performance compared to the U-Net model. Since it had the highest accuracy performance, it was picked for the onboard inference implementation. Its performance is presented in Figure 36.



Test_Batch (3 Images)	Keypoints RMS Error
Batch_1	2.4954
Batch_2	1.9872
Batch_3	1.7521
Batch_4	1.9512
Batch_5	2.2345
Batch_6	1.7896

Mean RMS Error (50 Images): 1.9814

Figure 36: Performance of ResNet34 - U-Net on Keypoint Localization.

From the preceding observations, the ResNet34 – U-Net model was chosen for the onboard inference implementation. Table 9 shows performance summary of the models that were investigated in this work.

Table 9: Summary of the Performance of Network Models.

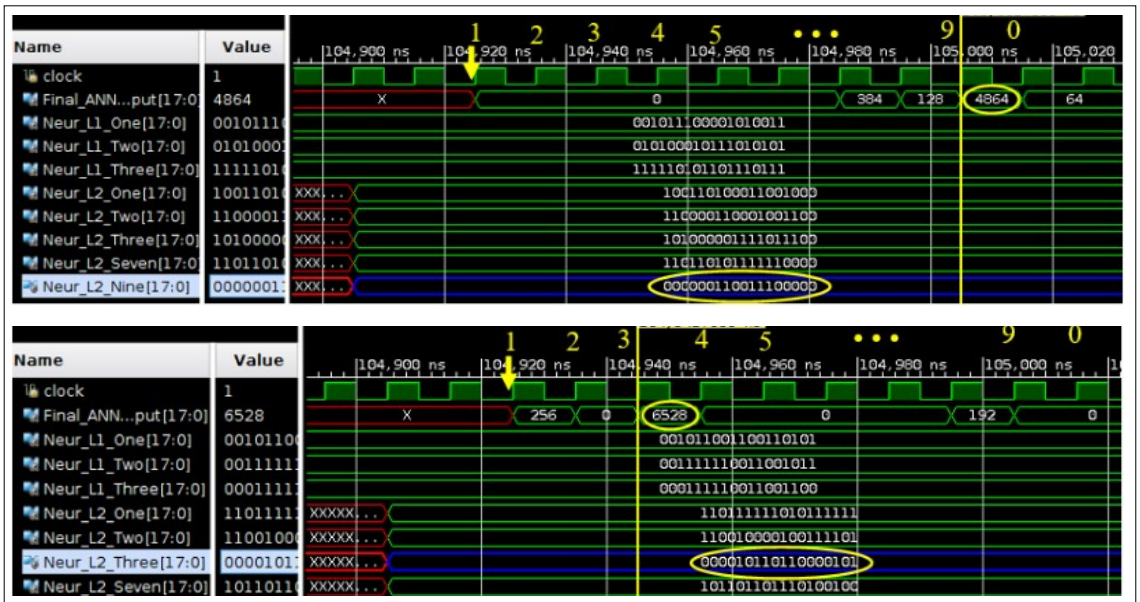
Approach	**Network Model**	**RMS-Error Performance**
Direct Regression on Full Image	ResNet-50	64.5214
Detection (Heatmap) on Full Image	U-Net	26.1183
Spacecraft Detection, Cropping & Heatmap-based Detection	U-Net	2.6371
Spacecraft Detection, Cropping & Heatmap-based Detection	ResNet34 - U-Net	1.9814

7.2 Results of FPGA Custom-based Inference

Though the main objective of this design was flexibility, modularity and parameterization, the network was tested both for accuracy and modularity. Two networks were implemented; one with 2 inputs, 10 hidden layer neurons and 4 output neurons and another targeting the MNIST dataset with 400 inputs, 25 hidden layer neurons and 10 output neurons. The same network topology and architecture is easily adoptable for different network sizes as long as the maximum DSP blocks and LUTs for the targeted FPGA are not exceeded. The modularized design allows seamless integration of different activation functions. It also allows combination of different implementations of the sigmoid per layer if necessary. Two gate-level sigmoid implementations were tested. These are the PLAN (H.Amin) [76] and SIG sigmoid (Tommiska) [77].

The performance of the MLP network on the MNIST dataset is shown in Fig. 37.The values circled in yellow show the current digit that is being predicted. The output layer has ten neurons with their values between 0 to 1 since they are sigmoid outputs. The data representation is 18 bit fixed point, but in the simulation the data is assumed to be 18 bit signed decimal. The predicted digit is the one with the highest value in the $Final\_ANN\_Output$ row. The number of clock cycles required by the network to process all the outputs is dependent on the number of neurons in each layer. This is because the output of one layer is pushed into the FIFO of next layer serially, i.e, single neuron output per clock cycle. The figure

also shows the results obtained with different activation function combinations for the 3-layer MLP network. Recognition of 20 digits was conducted, i.e. 2 samples for every digit [0 to 9]. Each digit is a 20 by 20 grid of pixels which is unrolled into a 400-dimensional vector.



	PLAN sigmoid in both layers	SIG sigmoid in both layers	PLAN in hidden layer and SIG in output layer	SIG in hidden layer and PLAN in output layer
Correct recognitions	20	15	19	17
Accuracy % (for 20 tested digits)	100	75	95	85

Figure 37: MLP Network Performance Simulation

From the preceding results, it is shown that combination of different implementations can help increase the accuracy of the network if one implementation results in poor accuracy. The FPGA resource utilization after synthesis (before optimization) for the modularized and parameterized MLP network is as shown in Table 10. This is the full network implementation including the activation function, control logic, processing elements and weights storage.

Table 10: FPGA Resource Utilization of 3-layer MLP Network.

	Resource	PLAN Sigmoid (Both Layers)	SIG Sigmoid (Both Layers)	PLAN(Hidden) & SIG(Output)	SIG(Hidden) & PLAN(Output)
DSP Block	DSP48E1 (%)	**35** (5.83)	**35** (5.83)	**35** (5.83)	**35** (5.83)
Block RAM	RAMB36E1	17	17	17	17
	RAMB18E1	27	27	27	27
	Total (%)	**30.5** (9.38)	**30.5** (9.38)	**30.5** (9.38)	**30.5** (9.38)
Slice LUTs	Logic	26198	26196	26190	26204
	Distr. RAM	120	120	120	120
	Total (%)	**26318** (25.95)	**26316** (25.95)	**26310** (25.95)	**26324** (25.96)
Slice Registers	Flip Flops	11853	11826	11852	11827
	Latch	513	498	513	498
	Total (%)	**12366** (6.10)	**12324** (6.08)	**12365** (6.10)	**12325** (6.08)

7.3 Results of FPGA IP-based Inference

The ResNet34-Unet model implemented in 4 had the highest accuracy as presented in 9. This network was implemented on the FPGA for onboard inference. In this final result section, the performance of the Ultra96v2-based inference and that of a PC-based inference is presented. The keypoint coordinates that were obtained from the onboard-inference were superimposed on those from the PC-inference. 100 test images were used for the inference testing phase. Figure 38 shows these results.

Table 11 presents the performance of the FPGA-based (8-bit quantized) inference as compared to PC-based (float) inference. It is shown that, although the inference on the MPSoC device is implemented with 8-bit quantization, there is no significant loss of accuracy from the PC-based implementation that uses 32-bit floats, with an average keypoint detection RMS error difference of less than 0.55. Hence, the hybrid Zynq MPSoC device is a viable candidate for implementing onboard inference in CNN-based spacecraft pose estimation algorithms.

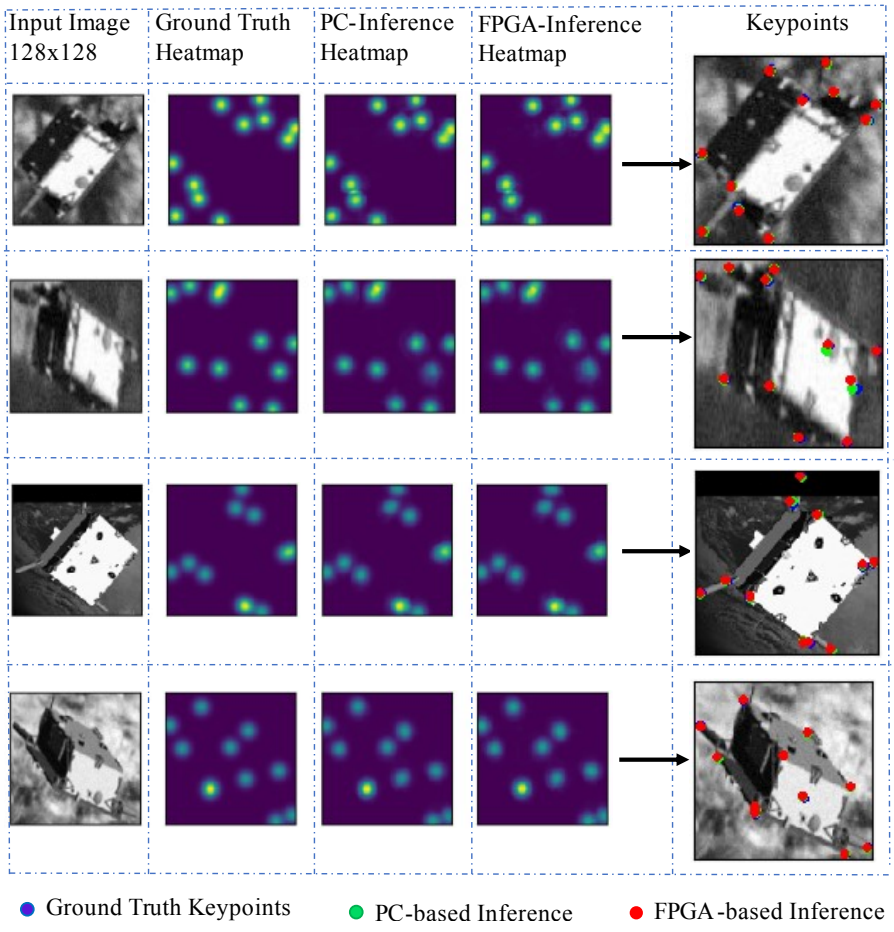


Figure 38: Performance of FPGA/MPSoC Inference vs. PC-based Inference.

Table 11: Performance of FPGA-based Inference vs PC-based.

		Keypoint Detection RMS-Error	
	Image	**PC-Inference**	**FPGA-Inference**
	Im9061	1.168	1.706
	Im9264	1.954	2.697
	Im9509	1.206	1.568
	Im9883	1.0	1.314
Average	(100 Images)	1.382	1.913

In addition to network RMS error performance, another key factor for consideration is the resource utilization of the FPGA fabric. The DPU is optimized for deep learning inference; hence, deep networks can be implemented on it without excessive resources. DPU resource utilization is dependent on its configuration parameters. The IP can be configured up to three cores. However, the target MPSoC device on the Ultra96v2 board is not sufficient for accommodating the corresponding high resource utilization. Only one core of the DPU was implemented in this work. Table 12 shows the resource utilization for different configurations of the DPU-based implementation.

Table 12: Deep Learning Processing Unit (DPU) Resource Utilization

	Configuration		
Resource	**B1600, 1 Core, High RAM Usage**	**B1152, 1 Core, High RAM Usage**	**B1024, 1 Core, Low RAM Usage**
BRAM	163.5	145	105.5
DSP Block	312	212	218
Slice LUTs	40,522	34,539	34,101
Flip Flops	61,595	49,451	49,519

Power consumption is another key factor to consider for onboard inference in space applications. Table 13 shows the on-chip power consumption estimates for different DPU configurations as estimated using the Report Power tool in Vivado IDE. This is an estimate of the total on-chip power consumed internally within the MPSoC device. It does not include power consumed by other off-chip devices and peripherals on the Ultra96v2 board. Figure 39 shows the evaluation board power consumption with no inference and during inference. It can be seen that the board consumes about 6.84 W when no inference job is running and about 8.16 W during inference. Hence, it can be concluded that the inference task consumes approximately 1.32 W. However, in a real mission, the MPSoC chip will be incorporated on a custom PCB with only the necessary peripherals as opposed to the full-blown circuit on the evaluation board used in this work. Therefore, the power consumption for such a custom board will be significantly less than that of the Ultra96v2 evaluation board.

Table 13: On-chip Power Consumption Estimates on Vivado IDE

Configuration	On-Chip Power (W)
B1600, 1 Core, High RAM Usage	3.9
B1152, 1 Core, High RAM Usage	3.3
B1024, 1 Core, Low RAM Usage	3.4

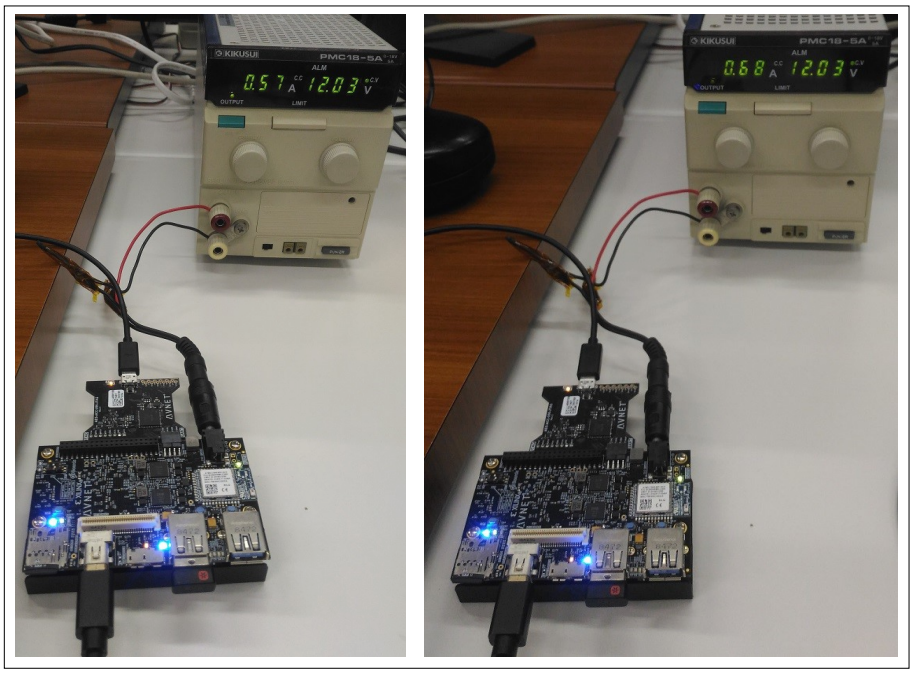


Figure 39: Power Consumption of Ultra96v2 Evaluation Board Outside and During Inference.

8 Discussion & Conclusions

This thesis work has presented various key areas pertaining to artificial intelligence for spacecraft autonomous operations. A comprehensive litertaure review on various machine and deep learning applications in the space field, especially onboard spacecraft, has been presented. In this thesis, the focus has been on FPGAs as a platform for onboard inference. In the first approach, a custom-based inference architecture was pursued. An MLP network that is modularized and parameterized was implemneted with the Kintex-7 FPGA as the target pltform. Such an architecure ANN is suitable for applications that can benefit from artificial intelligence but don't require the complexity and size of deep convolutional neural networks and other recent deep learning architectures. Thus the flexible modularized architecture presented here is efficient in adopting to different network sizes without much changes in the overall design.

In the second approach, IP-based, implementation of an FPGA-based onboard inferencing of CNN-based spacecraft pose estimation is presented. The focus of this approach is in accelerating the CNN part of such a pose estimation flow i.e., the landmark/keypoint localization for the 3D spacecraft structure. Once the landmarks have been localized, the rest of the pose estimation flow can be performed on the CPU part. Previous studies have focused on general PC-based implementation without exploring how such algorithms can be adapted to real-case applications. In this implementation, the Zynq MPSoC has been proposed as a suitable device for realizing onboard inference. A complete flow for such an implementation has been presented in this paper, beginning from network training, freezing, quantization, compilation, and, eventually, deployment to the board.

Direct regression and detection-based landmark localization were investigated. The former had poor performance when compared to the latter. Encoder-decoder model architectures were found to have better landmark localization performance. Of these, the ResNet34-U-Net model was found to have higher accuracy than a pure U-Net model. The performance was greatly enhanced by first detecting the spacecraft within the image. YOLOv3 was used for this detection and the image was cropped along the bounding box. The landmark localization network was then

fed the cropped image as input. This approach was found to have superior accuracy with an average RMS error of 1.7896 as compared to the uncropped approach performance of an average RMS error of 26.4587.

Xilinx UltraScale+ MPSoC was used as the target device. Its average on-chip power consumption of 3.5 W is low enough for power-limited spacecrafts and satellites. It should be noted that the evaluation board used in this work contains numerous extra peripherals and components that add to the overall power consumption. A custom design utilizing the MPSoC chip and off-chip memory would be suitable for a real space mission. Vivado was used for the hardware design i.e., programmable logic design. DPU IP core was utilized in order to implement the inference engine.It is configurable for various network types and supports most of the operations in typical CNN architectures. This is very convenient for adapting it to different spacecraft operations and not only pose estimation. It is shown that the onboard implementation accuracy is comparable to the PC-based inference, with an average RMS error difference of 0.531.

This work can further be developed in order to achieve an end-to-end FPGA-based spacecraft pose estimation. This will include interfacing to a real-time camera and performing the CPU-based algorithms for the final pose estimation tasks. The current implementation runs the two networks, YOLOv3 and ResNet34-U-Net, as standalone. The cropped images from the detection phase are stored in memory from which the DPU reads and runs them through the landmark localization phase. In an end-to-end implementation, these two phases would be coupled together. The inference time would be accurately measured in such an end-to-end implementation and it is estimated to be in tens of milliseconds that is consistent with other applications that have used Xilinx DPU in their designs.

Further, different inference engines for FPGA-based designs can be investigated. The DPU inference engine used in this work is a vendor IP, hence not portable to FPGAs from other vendors/manufacturers. Hence a more generic engine is desirable so that one is not locked into specific vendors. Moreso, investigation of better performance of other inference engines can be carried out. Another key area that can be studied further in this line of research is on the training data.

The ESA SPEED dataset was used for the CNN-based pose estimation problem. It would be desirable to investigate other datasets available for similar problems. This will help in studying the scalability of the developed methods and algorithms to spacecrafts of different models and images of varying quality.

The field of artificial intelligence is a fast-paced one. Hence, there will be new and more efficient hardware that can be utilized in onboard space applications. More so, Space 2.0 heralds new possibilities and applications in space. This could be in onboard AI for autonomous space exploration. Planetary and asteroid rovers would use AI to explore, map, and navigate terrain to avoid hazards, pick-up objects, and collect and analyze samples. On-orbit operations such as servicing, assembly and debris removal would also utilize robotic systems. These would also require significant increase in image processing, autonomous navigation, reading telemetry data from sensors, and controlling actuators. All of this information will have to be processed in real-time to enable remote space exploration. Hence the need for efficient hardware suitable for onboard inference in such applications will be a critical component in the future of space exploration.

Such hardware will inherently need to be computationally powerful with low power consumption. This would facilitate onboard inference for various spacecraft applications.Hybrid SoCs and FPGAs will play a dominant role in this area due to their inherent parallelism which is critical in deep neural networks inference. For space applications, such hardware would also need to be radiation hardened or tolerant. Towards this, in May 2020 Xilinx unveiled the Radiation Tolerant Kintex UltraScale FPGA targeting high throughput and bandwidth applications. Among its highlighted targeted applications is high performance edge inference in space. This underpins the importance of efficient hardware for onboard inference in future space applications that will increasingly employ autonomy and AI operations.

This work has resulted in the following conference and journal papers:

- Kiruki Cosmas, Kenichi Asami, "Utilization of FPGA for Onboard Inference of Landmark Localization in CNN-Based Spacecraft Pose Estimation" Aerospace 7, no. 11 (2020): 159.
- Kiruki Cosmas, Kenichi Asami, "Implementation of Machine Learning Meth-

ods on FPGA for Onboard Satellite Operation", 70th International Astronautical Congress (IAC), 21st – 25th Oct 2019, Washington DC, USA.

- Kiruki Cosmas, Kenichi Asami, "Flexible Modularized Artificial Neural Network Implementation on FPGA", IEEE 5th Intl. Conference on Soft Computing & Machine Intelligence, Nov 21-22, 2018, Nairobi, Kenya

Other academic papers published in the course of this work are included below:

- Kiruki Cosmas, "Arid and Semi-Arid Lands Satellite (ASAL-SAT): A LoRa ground sensor network for easing life in Sub-Saharan Africa ASAL areas", 6th UNISEC-Global Meeting, 19th Nov 2018, Strasbourg, France
- Fajardo, Isai, Aleksander A. Lidtke, Jesus Gonzalez-Llorente, Dmytro Faizullin et al. "Design, Implementation, and Operation of a Small Satellite Mission to Explore the Space Weather Effects in LEO" Aerospace 6, no. 10 (2019): 108.

9 Appendices

A. FPGA Verilog MLP Inference

Neural_Network.v

```
1 `timescale 1ns / 1ps
2 //
    //////////////////////////////////////////////////////////////////////////////////

3 // Company:
4 // Engineer:
5 //
6 // Create Date: 05/21/2018 05:11:55 PM
7 // Design Name:
8 // Module Name: Neural_Network
9 //
    //////////////////////////////////////////////////////////////////////////////////

10 //No_initial_Inputs should contain both the inputs and the Bias
    input of one
11
12 module Neural_Network #(parameter data_Width = 18, Frac=13,
    addr_Width = 14, depth= 11000,No_initial_Inputs=401,
    No_Neurons_HiddenLayerOne=25,No_Neurons_HiddenLayerTwo=10,
    No_Neurons_OutputLayer=4) (
13     input clock,rdFiFo,rdFiFoOutLyr,
14     input resetOrig,
15     output signed [data_Width-1:0]  MUX_output_Small,
    LyrTwoFIFOInputData,Final_ANN_Output,
16     output signed [data_Width-1:0] Neur_L1_One,Neur_L1_Two,
    Neur_L1_Three,Neur_L1_Four,Neur_L1_Five,//The aligned PE output
     to be fed into next layer
17     output signed [data_Width-1:0] Neur_L1_Six,Neur_L1_Seven,
    Neur_L1_Eight,Neur_L1_Nine,Neur_L1_Ten,
18     output signed [data_Width-1:0] Neur_L2_One,Neur_L2_Two,
    Neur_L2_Three,Neur_L2_Four,Neur_L2_Five,//same length as the
    inputs ie fixed point 18 bits
19     output signed [data_Width-1:0] Neur_L2_Six,Neur_L2_Seven,
    Neur_L2_Eight,Neur_L2_Nine,Neur_L2_Ten,//same length as the
```

```
    inputs ie fixed point 18 bits
20   output WeightsLoaded
21   );
22
23   //Internal Variables
24  wire dataReadyLayerOne;
25  wire dataReadyFinal,dataReadyHidden,enableFIFO_OutLyr,
    LyrTwoOperStart;
26
27  wire signed [data_Width-1:0] Neur_L1_Eleven,Neur_L1_Twelve,
    Neur_L1_Thirteen,Neur_L1_Fourteen,Neur_L1_Fifteen;
28  wire signed [data_Width-1:0] Neur_L1_Sixteen,Neur_L1_Seventeen,
    Neur_L1_Eighteen,Neur_L1_Nineteen,Neur_L1_Twenty;
29  wire signed [data_Width-1:0] Neur_L1_TwentyOne,
    Neur_L1_TwentyTwo,Neur_L1_TwentyThree,Neur_L1_TwentyFour,
    Neur_L1_TwentyFive;
30
31 //Get the MUX output back to the 12 bit size to avoid overflow in
    PE
32 //This already accomplished by fixed point PE
33 wire signed [data_Width-1:0] MUX_output;
34 assign MUX_output_Small=MUX_output;
35
36 //instantiate the Hidden Layer
37   HiddenLayerOne #(data_Width, Frac, addr_Width, depth,
    No_initial_Inputs,No_Neurons_HiddenLayerOne) myHiddenOne(
38   .clock          (clock),
39   .resetOrig      (resetOrig),
40   .dataReadyLyrOne (dataReadyLayerOne),//The FSm has given
    indication that layer one neurons have finished their
    calculations
41   .WeightsLoaded      (WeightsLoaded),
42   .Neur_L1_One    (Neur_L1_One),
43   .Neur_L1_Two    (Neur_L1_Two),
44   .Neur_L1_Three    (Neur_L1_Three),
45   .Neur_L1_Four    (Neur_L1_Four),
46   .Neur_L1_Five    (Neur_L1_Five),
47   .Neur_L1_Six    (Neur_L1_Six),
48   .Neur_L1_Seven    (Neur_L1_Seven),
```

```
49      .Neur_L1_Eight    (Neur_L1_Eight),
50      .Neur_L1_Nine    (Neur_L1_Nine),
51      .Neur_L1_Ten   (Neur_L1_Ten),
52      .Neur_L1_Eleven (Neur_L1_Eleven),
53      .Neur_L1_Twelve (Neur_L1_Twelve),
54      .Neur_L1_Thirteen   (Neur_L1_Thirteen),
55      .Neur_L1_Fourteen   (Neur_L1_Fourteen),
56      .Neur_L1_Fifteen    (Neur_L1_Fifteen),
57      .Neur_L1_Sixteen    (Neur_L1_Sixteen),
58      .Neur_L1_Seventeen  (Neur_L1_Seventeen),
59      .Neur_L1_Eighteen   (Neur_L1_Eighteen),
60      .Neur_L1_Nineteen   (Neur_L1_Nineteen),
61      .Neur_L1_Twenty     (Neur_L1_Twenty),
62      .Neur_L1_TwentyOne  (Neur_L1_TwentyOne),
63      .Neur_L1_TwentyTwo  (Neur_L1_TwentyTwo),
64      .Neur_L1_TwentyThree    (Neur_L1_TwentyThree),
65      .Neur_L1_TwentyFour     (Neur_L1_TwentyFour),
66      .Neur_L1_TwentyFive (Neur_L1_TwentyFive)
67      );
68
69
70  //Instantiate the MUX
71  //This MUX is linking the 10  neurons in layer one to the 4
        neurons in layer two
72  //The MUX will start loading data once the dataReady from previous
         layer is received. This should be done concurrently
73  //with the FIFO starting to read off the MUX outputs
74      MUX # (No_Neurons_HiddenLayerOne, data_Width) myMUX( //Ensure
        the Number of neurons in previous layer is correct
75      .clock              (clock),
76      .StartTrsferNxtLyr  (dataReadyLayerOne),//Layer One has
        finshed operations hence start transferring the neurons'
        outputs into the next layer
77      .resetPE            (resetOrig),//(ResetPE),
78      .NeuronOne          (Neur_L1_One),
79      .NeuronTwo          (Neur_L1_Two),
80      .NeuronThree        (Neur_L1_Three),
81      .NeuronFour         (Neur_L1_Four),
82      .NeuronFive         (Neur_L1_Five),
```

```
83      .NeuronSix          (Neur_L1_Six),
84      .NeuronSeven        (Neur_L1_Seven),
85      .NeuronEight        (Neur_L1_Eight),
86      .NeuronNine         (Neur_L1_Nine),
87      .NeuronTen          (Neur_L1_Ten),
88      .NeuronEleven       (Neur_L1_Eleven),
89      .NeuronTwelve       (Neur_L1_Twelve),
90      .NeuronThirteen     (Neur_L1_Thirteen),
91      .NeuronFourteen     (Neur_L1_Fourteen),
92      .NeuronFifteen      (Neur_L1_Fifteen),
93      .NeuronSixteen      (Neur_L1_Sixteen),
94      .NeuronSeventeen    (Neur_L1_Seventeen),
95      .NeuronEighteen     (Neur_L1_Eighteen),
96      .NeuronNineteen     (Neur_L1_Nineteen),
97      .NeuronTwenty       (Neur_L1_Twenty),
98      .NeuronTwentyOne    (Neur_L1_TwentyOne),
99      .NeuronTwentyTwo    (Neur_L1_TwentyTwo),
100     .NeuronTwentyThree  (Neur_L1_TwentyThree),
101     .NeuronTwentyFour   (Neur_L1_TwentyFour),
102     .NeuronTwentyFive   (Neur_L1_TwentyFive),
103     .MUX_output         (MUX_output)
104      );
105
106 //instantiate the Hidden Layer Two
107      HiddenLayerTwo #(data_Width,Frac, addr_Width, depth,
    No_Neurons_HiddenLayerOne,No_Neurons_HiddenLayerTwo)
    myHiddenTwo(
108     .clock                  (clock),
109     .resetOrig              (resetOrig),
110     .rdFiFo                 (rdFiFo),//Not used in the real
    operation
111     .dataReadyPrevLayer     (dataReadyLayerOne),
112     .MUX_Inputs             (MUX_output_Small), // to evaluate
    whether MUX is giving correct data & length
113     .Neur_L2_One            (Neur_L2_One),
114     .Neur_L2_Two            (Neur_L2_Two),
115     .Neur_L2_Three          (Neur_L2_Three),
116     .Neur_L2_Four           (Neur_L2_Four),
117     .Neur_L2_Five           (Neur_L2_Five),
```

```
118       .Neur_L2_Six            (Neur_L2_Six),
119       .Neur_L2_Seven          (Neur_L2_Seven),
120       .Neur_L2_Eight          (Neur_L2_Eight),
121       .Neur_L2_Nine           (Neur_L2_Nine),
122       .Neur_L2_Ten            (Neur_L2_Ten),
123       .dataReadyLyrTwo        (dataReadyFinal),//(dataReadyHidden)
124       .LyrTwoFIFOInputData    (LyrTwoFIFOInputData)// To evaluate
      whether layer two is receiving correct data from the MUX (Layer
       one outputs after sigmoi)
125       );
126
127 //instantiate the OutputMUX
128     OutputMUX_n_Sigmoid # (No_Neurons_HiddenLayerOne, data_Width)
      myOutputMUX( //Ensure the Number of neurons in previous layer
      is correct
129      .clock               (clock),
130      .StartTrsferNxtLyr   (dataReadyFinal),//Output Layer has
      finshed operations hence pass the outputs to the sigmoid for
      the final data output
131      .resetPE             (resetOrig),//(ResetPE),
132      .NeuronOne           (Neur_L2_One),
133      .NeuronTwo           (Neur_L2_Two),
134      .NeuronThree         (Neur_L2_Three),
135      .NeuronFour          (Neur_L2_Four),
136      .NeuronFive          (Neur_L2_Five),
137      .NeuronSix           (Neur_L2_Six),
138      .NeuronSeven         (Neur_L2_Seven),
139      .NeuronEight         (Neur_L2_Eight),
140      .NeuronNine          (Neur_L2_Nine),
141      .NeuronTen           (Neur_L2_Ten),
142      .MUX_output          (Final_ANN_Output)
143     );
144
145 endmodule
```

BRAM_N_Controller.v

```
1 `timescale 1ns / 1ps
2 //
      //////////////////////////////////////////////////////////////////////////////////
```

```
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 05/15/2018 02:05:00 PM
7  // Design Name:
8  // Module Name: BRAM_N_Controller
9  // Project Name:
10 //
   //////////////////////////////////////////////////////////////////////////////////
11
12
13 module BRAM_N_Controller #(parameter data_Width = 12, addr_Width =
       14, depth= 11000,No_Neurons_PreviousLayer=3,
   Num_of_Neurons_CurrentLayer=3) (
14     input clock,
15     input reset,BRAM_Enable,
16     output reg [addr_Width:0] CountVal,CounterInputs,//worst case
   scenario should be hardcoded
17     output reg signed [data_Width-1:0] dataOut,dataInputOut,
18     output reg WeightsLoaded
19     );
20        //Internal variables
21     reg signed [data_Width-1:0] Inputs_PrevLayer [0:
   No_Neurons_PreviousLayer];//The column of inputs from the
   previous layer
22                                                 //This is for the
       input layer whose inputs are the input and the bias input at
   index 0
23     reg [addr_Width:0] max_count= No_Neurons_PreviousLayer*
   Num_of_Neurons_CurrentLayer + 1;
24    // reg [addr_Width:0] max_count= (No_Neurons_PreviousLayer+1)*
   Num_of_Neurons_CurrentLayer + 1;//add one to accomodate the
   BIAs for each neuron
25     reg signed [data_Width-1:0]   Input_data=0,
   Inputs_PrevLayerData=0;//decoy for BRAM to force XST to infer
   BRAM
26   //   reg [addr_Width:0] CounterInputs;
```

```
27
28          //Infer the RAM ie RAM modelling
29      reg signed [data_Width-1:0] RAM [0:depth-1];
30
31
32      //Initialize the RAM contents to any value if necessary.
       Default is 0
33          initial
34      begin
35     //   $readmemb("sine_values.dat",RAM,0,depth-1);
36     //   $readmemb("InputData.dat",Inputs_PrevLayer,0,
       No_Neurons_PreviousLayer);
37      $readmemb("MyWeightsOneFxdPointBinary.dat",RAM,0,depth-1);//
       Weights plus the bias combined
38      $readmemb("MyInputsExpsFxdPBinary.dat",Inputs_PrevLayer,0,
       No_Neurons_PreviousLayer);//Input data
39      WeightsLoaded<=0;
40      end
41
42      always @ (posedge clock)
43      begin : Counter
44
45      if (reset) begin
46      CountVal<=0;
47      WeightsLoaded=0;//This is to disable the Neuron when an
       original reset is issued again
48                      //This is useful when processing another
       example (M) in the ANN
49                      //The neurons will be reset now and only
       enabled when they have been loaded afresh with the weights in
       BRAM
50                      //This is necessary because the FiFo are
       currently empty after processing the current Example (M)
51      end
52
53      else if (CountVal<max_count) begin
54
55          if (BRAM_Enable) begin //decoy for synthesis tool to use
       BRAM but its still using Distributed RAM
```

```
56              RAM [CountVal] <= Input_data;
57              dataOut <= RAM [CountVal];
58          end
59
60          else begin
61          dataOut <= RAM [CountVal];
62          CountVal<=CountVal+1;
63          end
64
65      end
66
67      else if (CountVal>=max_count)//All weights for the current
    layer have been loaded
68      WeightsLoaded<=1;
69
70      end
71
72      //Load the inputs from BRAM to the FiFO. In the real
    implementation this will be different
73      //This is just for the first layer ie hidden layer one
74      //For the other hidden layers, the inputs will be picked from
    the MUX linking the two hidden layers
75      always @ (posedge clock) begin
76      if (reset) begin
77      CounterInputs<=0;
78      end
79
80       else if (CounterInputs<=No_Neurons_PreviousLayer+1) begin
81
82              if (BRAM_Enable) begin //decoy for synthesis tool to
    use BRAM but its still using Distributed RAM
83                  Inputs_PrevLayer [CounterInputs] <=
    Inputs_PrevLayerData;
84                  dataInputOut <= Inputs_PrevLayer [CounterInputs];
85              end
86
87              else begin
88              dataInputOut <= Inputs_PrevLayer [CounterInputs];
89              CounterInputs<=CounterInputs+1;
```

```
            end

        end

    end

endmodule
```

TommiskaSigmoid.v

```
`timescale 1ns / 1ps
//
    //////////////////////////////////////////////////////////////////////////////////

// Company:
// Engineer:
//
// Create Date: 06/15/2018 11:49:00 PM
// Design Name:
// Module Name: TommiskaSigmoid
//
    //////////////////////////////////////////////////////////////////////////////////


module TommiskaSigmoid # (parameter dataWidth=9,data_WidthSig=18)(
input signed [dataWidth-1:0] inputData,
output Z_One,Z_Two,Z_Three,Z_Four,signbit,
output signed [dataWidth-1:0] sigOutMinusMUX,Y_Complement,
    sigOutMUX,//9 bits
output signed [dataWidth-1:0] Y_MUX_One,Y_MUX_Final,
output [17:0] sigmoidOutput
    );

    wire x_0,x_1,x_2,x_3,x_4,x_5; // x_0 is the LSB
    wire p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9,p_10;
    wire p_11,p_12,p_13,p_14,p_15,p_16,p_17,p_18,p_19,p_20;
    wire p_21,p_22,p_23,p_24,p_25,p_26,p_27;
    wire y_0,y_1,y_2,y_3,y_4,y_5,y_6;

```

```
26      wire [6:0] sigOut; // 7 fractional bits
27      wire [6:0] sigOutMinus; //1-Y
28
29      //obtain the sign bit
30      assign signbit=inputData[dataWidth-1];
31
32     //Pad zeros to the 7 bits to make it 9 bits like the
       SigmoidPLAN for easy integration to overall network and MUX
33     //These will be the inputs to the MUX
34      assign sigOutMUX[0] = 0;// no 4th fractional bit
35      assign sigOutMUX[1] = sigOut[0];
36      assign sigOutMUX[2] = sigOut[1];
37      assign sigOutMUX[3] = sigOut[2];
38      assign sigOutMUX[4] = sigOut[3];
39      assign sigOutMUX[5] = sigOut[4];
40      assign sigOutMUX[6] = sigOut[5];
41      assign sigOutMUX[7] = sigOut[6];
42      assign sigOutMUX[8] = 0; //No 4th integer bit
43
44      assign sigOutMinusMUX[0] = 0;// no 4th fractional bit
45      assign sigOutMinusMUX[1] = sigOutMinus[0];
46      assign sigOutMinusMUX[2] = sigOutMinus[1];
47      assign sigOutMinusMUX[3] = sigOutMinus[2];
48      assign sigOutMinusMUX[4] = sigOutMinus[3];
49      assign sigOutMinusMUX[5] = sigOutMinus[4];
50      assign sigOutMinusMUX[6] = sigOutMinus[5];
51      assign sigOutMinusMUX[7] = sigOutMinus[6];
52      assign sigOutMinusMUX[8] = 0; //No 4th integer bit
53      //The input is positive number with 7 bits: 1 sign bit, 3
       integer bits and 3 fractional bits
54      //But the MSB ie the sign bit is not used.
55      //Y_MUX_One is 9 bits. Discard MSB ie the signbit and
       Y_MUX_One[1] and pick the next 6 bits
56     assign x_0 = Y_MUX_One[dataWidth-8];
57     assign x_1 = Y_MUX_One[dataWidth-7];
58     assign x_2 = Y_MUX_One[dataWidth-6];
59     assign x_3 = Y_MUX_One[dataWidth-5];
60     assign x_4 = Y_MUX_One[dataWidth-4];
61     assign x_5 = Y_MUX_One[dataWidth-3];
```

```
62    //assign x_5 = Y_MUX_One[dataWidth-2]; //discard the MSB
      integer bit
63    //assign x_6 = inputData[dataWidth-1];  //redundant sign bit
64
65    //Input
66     and g1 (p_1,x_2,x_5);
67     and g2 (p_2,x_5,x_4);
68     and g3 (p_3,x_5);
69     and g4 (p_4,x_5,x_3);
70     and g5 (p_5,x_4,!x_3,!x_2,!x_1,!x_0);
71     and g6 (p_6,!x_4,x_3,!x_2,!x_1,!x_0);
72     and g7 (p_7,!x_4,!x_3,x_2,x_1,!x_0);
73     and g8 (p_8,x_3,!x_2,x_1,!x_0);
74     and g9 (p_9,x_4,x_3,x_1,x_0);
75     and g10 (p_10,x_4,!x_3,x_1,x_0);
76     and g11 (p_11,x_4,x_2,x_1);
77     and g12 (p_12,!x_4,x_3,x_1,x_0);
78     and g13 (p_13,x_3,x_2,x_1);
79     and g14 (p_14,x_3,!x_1,x_0);
80     and g15 (p_15,x_4,x_2,x_0);
81     and g16 (p_16,x_4,!x_3,!x_2,x_1);
82     and g17 (p_17,!x_4,!x_3,!x_2,x_1);
83     and g18 (p_18,!x_4,x_3,x_2);
84     and g19 (p_19,x_4,x_3,x_2);
85     and g20 (p_20,x_3,x_2);
86     and g21 (p_21,!x_4,x_2,x_1,x_0);
87     and g22 (p_22,!x_4,x_2,!x_1,x_0);
88     and g23 (p_23,x_4,!x_3,x_2,!x_1);
89     and g24 (p_24,x_4,!x_2,!x_1,x_0);
90     and g25 (p_25,!x_4,!x_3,!x_2,x_0);
91     and g26 (p_26,x_4,x_3);
92     and g27 (p_27,x_4,x_3,!x_2,!x_0);
93
94     //output
95    // assign y_6=1;
96     or g28 (y_5,p_3,p_5,p_8,p_10,p_11,p_12,p_13,p_14,p_15,p_16,
      p_18,p_23,p_24,p_26);
97     or g30 (y_4,p_3,p_5,p_6,p_10,p_11,p_15,p_16,p_20,p_24,p_26);
98     or g31 (y_3,p_3,p_6,p_11,p_13,p_17,p_18,p_21,p_26);
```

```
99      or g32 (y_2,p_3,p_6,p_7,p_9,p_12,p_13,p_16,p_19,p_23,p_25);
100     or g33 (y_1,p_3,p_6,p_7,p_8,p_12,p_21,p_22,p_23,p_24,p_27);
101     or g34 (y_0,p_1,p_2,p_4,p_5,p_7,p_8,p_10,p_13,p_14,p_15,p_18,
        p_22);
102
103     //assign the output
104     assign sigOut[6] = 1; //MSB
105     assign sigOut[5] = y_5;
106     assign sigOut[4] = y_4;
107     assign sigOut[3] = y_3;
108     assign sigOut[2] = y_2;
109     assign sigOut[1] = y_1;
110     assign sigOut[0] = y_0;//LSB
111
112     Tommiska_2s_Complement  # (dataWidth) mytwoComp(
113     .inputData          (inputData),
114     .Yout               (Y_Complement)       //the 2's complement
115     );
116
117     Multiplexer_8bit # (dataWidth) myMUXOne  (
118     .signbit            (signbit),
119     .inputA             (Y_Complement),  //the 2's complement
120     .inputB             (inputData),    //original data
121     .Ysel               (Y_MUX_One) //9 bit to feed into the
        SigmoidFunct
122     );
123
124     //Instantiate the subtractor for 1-Y
125     fullSubtractor # (7)  mySub(  //The data length is the 7
        fractional bits
126     .inputData      (sigOut),
127     .Yout           (sigOutMinus)
128     );
129
130   //Instantiate the Final MUX for the final output
131     Multiplexer_8bit # (dataWidth) myMUXFinal  (
132   .signbit            (signbit),
133   .inputA             (sigOutMinusMUX),  //1-Y for negative
      numbers
```

```
134     .inputB             (sigOutMUX),    //Y
135     .Ysel               (Y_MUX_Final) //9 bit final output to feed
        into the next layer
136     );
137 
138   //The network neuron inputs are 18 bits: 1 sign, 4 integer and
        13 fractional bits
139   //All the bits for Tommiska Sigmoid Output are 7 fractional bits
140     assign sigmoidOutput[data_WidthSig-1:data_WidthSig-5]=0;//sign
        bit always 0 and all the 4 integer bits are 0
141     assign sigmoidOutput[data_WidthSig-6]=Y_MUX_Final[7]; //MSB
142     assign sigmoidOutput[data_WidthSig-7]=Y_MUX_Final[6];
143     assign sigmoidOutput[data_WidthSig-8]=Y_MUX_Final[5];
144     assign sigmoidOutput[data_WidthSig-9]=Y_MUX_Final[4];
145     assign sigmoidOutput[data_WidthSig-10]=Y_MUX_Final[3];
146     assign sigmoidOutput[data_WidthSig-11]=Y_MUX_Final[2];
147     assign sigmoidOutput[data_WidthSig-12]=Y_MUX_Final[1]; //LSB
148     assign sigmoidOutput[data_WidthSig-13:0]=0;//rest of the 6
        fractional bits
149 
150 
151 endmodule
```

B. Data Preparation & Preprocessing for Keypoints Localization

combine_dataframes.py

```
1  import numpy as np
2  import os
3  import pandas as pd
4  import csv
5 
6  def main():
7    #df = pd.read_csv('image_index_AllFinal.txt', index_col=False,
       quoting=csv.QUOTE_NONE)
8    #df = pd.read_csv('image_index_coord.txt', index_col=False,
       quoting=csv.QUOTE_NONE)
9    #df = pd.read_csv('exp_kpt_indices.txt', index_col=False,
       quoting=csv.QUOTE_NONE)
10   df = pd.read_csv('keypoints_coord_withIndices.csv', index_col=
```

```
    False, quoting=csv.QUOTE_NONE)
11  df.set_index('Indices', inplace = True) #Make the indices as the
      row index
12
13  #df_two = pd.read_csv('combineAll_samples.txt', index_col=False)
       #Image coordinates 12K images
14  #df_two = pd.read_csv('coord_extracted_headers.csv', index_col=
     False)  #Image coordinates 12K images
15  df_two = pd.read_csv('image_coord_withIndices.csv', index_col=
     False)  #Image coordinates 12K images
16  df_two.set_index('Indices', inplace = True)
17
18  #merge with both indices(python pandas index) true i.e. only
     data with common indices will be preserved
19  #df_three = pd.merge(df, df_two, left_index = True, right_index
     = True)
20  #df_three = pd.merge(df, df_two, left_index = True, right_index
     = True)
21  df_three = pd.merge(df, df_two, left_index = True, right_index =
      True)
22
23  #df_three.to_csv(r'keypoints_coord_withIndices.csv', header=True
     , sep = ',', index=None, quoting=csv.QUOTE_NONE)
24  #df_three.to_csv(r'exp_kpts_images.txt', header=True, sep = ',',
      index=None, quoting=csv.QUOTE_NONE)
25  df_three.to_csv(r'Keypoints_Images.csv', header=True, sep = ',',
      index=None, quoting=csv.QUOTE_NONE)
26
27
28 if __name__ == '__main__':
29     main()
```

extract_coordinates.py

```
1 import ast
2 import numpy as np
3 import pandas as pd
4
5 #input file
6 with open('coordinates_one.txt', 'r') as fin:
```

```
7   newArray = np.zeros((11,2))  #There are 11 keypoints with 2
     points each i.e. x & y coord
8   data = []
9   image_scale = 224
10  x_factor = 1920./224
11  y_factor = 1200./224
12
13  for line in fin:
14    newDict = ast.literal_eval(line)
15    coord = newDict["joints"]  #List of coordinates i.e. List if
     Lists
16
17    #Lets scale the keypoints to match the scaled image size
18    for index, keypt in enumerate(coord):
19      newArray[index][0] = keypt[0] / x_factor
20      newArray[index][1] = keypt[1] / y_factor
21    '''
22    # Let's flatten the list
23    flattened = []
24    for sublist in coord:
25      for val in sublist:
26        flattened.append(val)
27    '''
28    #The above can be written succintly using list comprehension
29    #This is faster than the unraveled loop & it avoids the append
     calls
30    #flattened = [val for sublist in coord for val in sublist]
31    flattened = [val for sublist in newArray for val in sublist]
32
33    #Instead of writing each line to disk, first accumulate data
     in a python data structure e.g. array
34    data.append(flattened)
35
36  #Write the accumulated data to a dataframe
37  df = pd.DataFrame(data)
38  df.to_csv(r'coord_extracted.csv', header=False, index=None, sep=
     ',', mode='a')
```

resize_images.py

```
import json
from scipy import ndimage, misc
from PIL import Image
from pylab import *
import numpy as np
import os
import pandas as pd
np.set_printoptions(threshold=sys.maxsize)

def main():
  #path = "C:\\Users\\cosma\\Google Drive\\Synced Folder_Academia
   \\Post-Midterm\\prepare_data_yolo\\ResNet_and_Bounding\\images"
  path = "E:\\DPU_downloads\\networks\\speed\\images\\train\\
   sample_12"
  #fout= open("imagearray.txt", "wt")
  size = 224, 224
  data = []
  # iterate through the names of contents of the folder
  for image_path in os.listdir(path):
    input_path = os.path.join(path, image_path)
    orig_im = Image.open(input_path)
    resizedImage  = orig_im.resize(size)
    array_resized = array(resizedImage)
    pix_val_list = list(resizedImage.getdata())
    pix_val_array = np.array(pix_val_list)
    data.append(pix_val_array)

  #print(data)
  df = pd.DataFrame(data)
  df.to_csv(r'sample_12.txt', header=False, index=None, sep=' ',
   mode='a')
  print('Number of columns in df:')
  print(len(df.columns))
  print('Number of rows in df:')
  print(len(df))
  #fout.write(str(pix_val))
  #fout.close()

    # pil_img = image.load_img('images/img000001.jpg', target_size
```

```
    =(1920, 1200))
37      # x = image.img_to_array(pil_img)
38      #print('The length of this image is'+ str(len(x)))
39
40  if __name__ == '__main__':
41      main()
```

save_images.py

```
1  import cv2 as cv
2  import pandas as pd
3  import numpy as np
4  import os
5
6  train_dir = "train_Images"
7  df_train = pd.read_csv('Keypoints_Images.csv')
8
9  n_train = df_train['Images'].size
10
11 print(f"Number of Training Images: {n_train}")
12
13 def save_images(arr, name, dir):
14   for i, image in enumerate(arr):
15     image = np.fromstring(image, dtype=int, sep=' ')
16     image = np.reshape(image, newshape=(224, 224))
17
18     fullname = name + str(i) + ".png"
19
20     cv.imwrite(os.path.join(train_dir,fullname), image)
21
22 train_arr = df_train['Images'].tolist()
23 save_images(train_arr, name="train", dir=train_dir)
```

scale_coordinates.py

```
1  import ast
2  import numpy as np
3
4  #input file
5  fin= open("coordinates_one.txt", "rt")
6  fout= open("coordinates_scaled.txt", "wt")
```

```
7
8 newArray = np.zeros((11,2))
9
10 for line in fin:
11   newDict = ast.literal_eval(line)
12   coord = newDict["joints"]
13   for index, keypt in enumerate(coord):
14     newArray[index][0] = (keypt[0] - 960) / 960  # Scale to (-1,1)
15     newArray[index][1] = (keypt[1] - 600) / 600  # Scale to (-1,1)
16   fout.write(str(newDict["image"]) + " " + str(newArray) +"\n")
17 fin.close()
18 fout.close()
```

C. Training for Keypoints Localization

unet_model.py

```
import numpy as np
import math
import matplotlib.pyplot as plt
import cv2 as cv

import keras.utils
from keras.callbacks import EarlyStopping, CSVLogger,
    ModelCheckpoint, ReduceLROnPlateau
from keras.layers import Conv2D, MaxPooling2D, Conv2DTranspose,
    concatenate, Reshape
from keras.models import Input, Model
from keras.optimizers import SGD, RMSprop, Adam
from keras.regularizers import l2

Nkeypoints = 11
W = 224
H = 224

log_dir = 'unet_logs/'

def UNET(input_shape):
    def downsample_block(x, block_num, n_filters, pooling_on=True)
    :

        x = Conv2D(n_filters, kernel_size=(3, 3), strides=1,
    padding='same', activation='relu',
                   name="Block" + str(block_num) + "_Conv1")(x)
        x = Conv2D(n_filters, kernel_size=(3, 3), strides=1,
    padding='same', activation='relu',
                   name="Block" + str(block_num) + "_Conv2")(x)
        skip = x

        if pooling_on is True:
            x = MaxPooling2D(pool_size=(2, 2), strides=2, padding=
    'valid', name="Block" + str(block_num) + "_Pool1")(x)

```

```
31          return x, skip
32
33      def upsample_block(x, skip, block_num, n_filters):
34
35          x = Conv2DTranspose(n_filters, kernel_size=(2, 2), strides
    =2, padding='valid', activation='relu',
36                              name="Block" + str(block_num) + "
    _ConvT1")(x)
37          x = concatenate([x, skip], axis=-1, name="Block" + str(
    block_num) + "_Concat1")
38          x = Conv2D(n_filters, kernel_size=(3, 3), strides=1,
    padding='same', activation='relu',
39                     name="Block" + str(block_num) + "_Conv1")(x)
40          x = Conv2D(n_filters, kernel_size=(3, 3), strides=1,
    padding='same', activation='relu',
41                     name="Block" + str(block_num) + "_Conv2")(x)
42
43          return x
44
45      input = Input(input_shape, name="Input")
46
47      # downsampling
48      x, skip1 = downsample_block(input, 1, 64)
49      x, skip2 = downsample_block(x, 2, 128)
50      x, skip3 = downsample_block(x, 3, 256)
51      x, skip4 = downsample_block(x, 4, 512)
52      x, _ = downsample_block(x, 5, 1024, pooling_on=False)
53
54      # upsampling
55      x = upsample_block(x, skip4, 6, 512)
56      x = upsample_block(x, skip3, 7, 256)
57      x = upsample_block(x, skip2, 8, 128)
58      x = upsample_block(x, skip1, 9, 64)
59
60      output = Conv2D(11, kernel_size=(1, 1), strides=1, padding='
    valid', activation='linear', name="output")(x)
61      output = Reshape(target_shape=(H*W*Nkeypoints,1))(output)
62
63      model = Model(inputs=input, outputs=output, name="Output")
```

```
64
65      return model
```

resnet34_unet.py

```
1  import numpy as np
2  import math
3  import matplotlib.pyplot as plt
4  import cv2 as cv
5
6  import keras.utils
7  import keras
8  from keras.callbacks import EarlyStopping, CSVLogger,
       ModelCheckpoint, ReduceLROnPlateau
9  from keras.layers import Activation, Add, BatchNormalization,
       Conv2D, UpSampling2D, concatenate, Reshape
10 from keras.models import Input, Model
11 from keras.optimizers import SGD, RMSprop, Adam
12 from keras.regularizers import l2
13
14 from classification_models.keras import Classifiers
15
16 log_dir = 'unet_logs/21st_Sept/'
17 im_size = 128
18 keypoints = 11
19 H=128
20 W=128
21
22 ResNet34, preprocess_input = Classifiers.get('resnet34')
23 Resmodel = ResNet34((im_size,im_size,1), weights=None)
24
25 '''
26 print(Resmodel.summary())
27 print(Resmodel.get_layer(index = 5).name)
28 print(Resmodel.layers[5].output_shape)
29
30 print(Resmodel.get_layer(index = 37).name)
31 print(Resmodel.layers[37].output_shape)
32
33 print(Resmodel.get_layer(index = 74).name)
```

```
print(Resmodel.layers[74].output_shape)
'''

def ConvBlock(X,channel,kernel_size,bn=True):
  x=Conv2D(filters=channel,kernel_size=(kernel_size,kernel_size),
   strides=(1,1),dilation_rate=(1,1),padding='SAME',
   kernel_initializer='he_normal')(X)
  if bn:
    x=BatchNormalization()(x)
  x=Activation('relu')(x)

  x=Conv2D(filters=channel,kernel_size=(kernel_size,kernel_size),
   strides=(1,1),dilation_rate=(1,1),padding='SAME',
   kernel_initializer='he_normal')(x)
  if bn:
    x=BatchNormalization()(x)
  x=Activation('relu')(x)
  return x

def unet_ResNet(output_channel):
  inp=Resmodel.input    #128x128x1

  skip1=Resmodel.layers[5].output #64x64x64
  skip2=Resmodel.layers[37].output #32x32x64
  skip3=Resmodel.layers[74].output #16x16x128
  skip4=Resmodel.layers[129].output #8x8x256
  encoder_final=Resmodel.layers[157].output #4x4x512

  #upsample
  filters=256
  k=1

  x=UpSampling2D()(encoder_final) #returns 8x8x512
  x=concatenate([x,skip4], name="Concat1") #returns 8x8x(512+256)
   =768
  x=ConvBlock(x,channel=filters,kernel_size=3) #returns 8x8x256
  filters //=2

  x=UpSampling2D()(x) #returns 16x16x256
```

```
68    x=concatenate([x,skip3], axis=-1, name="Concat2") #returns 16
       x16x384
69    x=ConvBlock(x,channel=filters,kernel_size=3) #returns 16x16x128
70    filters //=2
71
72    x=UpSampling2D()(x) #returns 32x32x128
73    x=concatenate([x,skip2], axis=-1, name="Concat3") #returns 32
       x32x192
74    x=ConvBlock(x,channel=filters,kernel_size=3) #returns 32x32x64
75    filters //=2
76
77    x=UpSampling2D()(x) #returns 64x64x64
78    x=concatenate([x,skip1], axis=-1, name="Concat4") #returns 64
       x64x128
79    x=ConvBlock(x,channel=filters,kernel_size=3) #returns 64x64x32
80    filters //=2
81
82    x=UpSampling2D()(x) #returns 128x128x32
83    x=ConvBlock(x,channel=filters,kernel_size=3) #returns 128x128x16
84    x =Conv2D(output_channel, kernel_size= (1,1), strides=(1,1),
       padding= 'same')(x)  #returns 128x128xkeypoints
85    x=Activation('sigmoid')(x)
86    x = Reshape(target_shape=(H*W*keypoints,1))(x)
87    model=Model(inputs=inp,outputs=x)
88
89    return model
```

training_keypoints.py

```
1  def create_callbacks(wts_fn, csv_fn, patience=5, enable_save_wts =
       True):
2
3      cbks = []
4
5      # early stopping
6      #early_stopper = EarlyStopping(monitor='val_loss', patience=
      patience)
7      #cbks.append(early_stopper)
8
9      # model checkpoint
```

```
    if enable_save_wts is True:
        model_chpt = ModelCheckpoint(filepath=wts_fn,
                                     monitor='val_loss',
                                     verbose=1,
                                     save_weights_only=True,
                                     save_best_only=True,
                                     period=2)

        cbks.append(model_chpt)

    # csv logger
    csv_logger = CSVLogger(csv_fn)
    cbks.append(csv_logger)

    return cbks

def trainModel(model, loss_type, n_epochs, old_lr, new_lr,
    train_gen, val_gen, load_saved_wts = False):

    if load_saved_wts is True:
        #wts_fn = log_dir + "lr=" + str(old_lr) + ".h5"
        wts_fn = "unet_logs/9th_Sept/lr=0.0001.h5"  #Latest
    weights
        model.load_weights(wts_fn)

    wts_fn = log_dir + "lr=" + str(new_lr) + ".h5"
    csv_fn = log_dir + "lr=" + str(new_lr) + ".csv"
    cbks = create_callbacks(wts_fn, csv_fn)

    optim = RMSprop(lr=new_lr)

    model.compile(loss="mean_squared_error", optimizer=optim,
    metrics=None)
    model.fit_generator(generator=train_gen,
                        validation_data=val_gen,
                        epochs=n_epochs,
                        callbacks=cbks)
    return model

```

```
46 def maskToKeypoints(mask):
47     # mask = np.reshape(mask, newshape=(96,96))
48     kp = np.unravel_index(np.argmax(mask, axis=None), dims=(
   im_size,im_size))
49     return kp[1], kp[0]
50 
51 def findCoordinates(mask):
52 
53     hm_sum = np.sum(mask)
54 
55     index_map = [j for i in range(im_size) for j in range(im_size)
   ]
56     index_map = np.reshape(index_map, newshape=(im_size,im_size))
57 
58     x_score_map = mask * index_map / hm_sum
59     y_score_map = mask * np.transpose(index_map) / hm_sum
60 
61     px = np.sum(np.sum(x_score_map, axis=None))
62     py = np.sum(np.sum(y_score_map, axis=None))
63 
64     return px, py
65 
66 def calcKeypoints(model, gen):
67     kps_gt = []
68     kps_preds = []
69     nbatches = len(gen)
70 
71     for i in range(nbatches+1):
72         # print("\nBatch {}".format(i))
73         imgs, batch_gt = gen[i]
74         batch_preds = model.predict_on_batch(imgs)
75         n_imgs = imgs.shape[0]
76         # print("\t# of Images {}".format(n_imgs))
77         for j in range(n_imgs):
78             mask_gt = batch_gt[j]
79             mask_gt = np.reshape(mask_gt, newshape=(im_size,
   im_size, 11))
80             mask_pred = batch_preds[j]
81             mask_pred = np.reshape(mask_pred, newshape=(im_size,
```

```
    im_size, 11))
82              nchannels = mask_gt.shape[-1]
83              # print(nchannels)
84              gt_list = []
85              pred_list = []
86
87              for k in range(nchannels):
88                  xgt, ygt = maskToKeypoints(mask_gt[:, :, k])
89                  xpred, ypred = maskToKeypoints(mask_pred[:, :, k])
90
91                  #xgt, ygt = findCoordinates(mask_gt[:, :, k]) #
    maskToKeypoints(mask_gt[:, :, k])
92                  #xpred, ypred = findCoordinates(mask_pred[:, :, k
    ])  # maskToKeypoints(mask_pred[:, :, k])
93
94                  gt_list.append(xgt)
95                  gt_list.append(ygt)
96
97                  pred_list.append(xpred)
98                  pred_list.append(ypred)
99
100             kps_gt.append(gt_list)
101             kps_preds.append(pred_list)
102
103     return np.array(kps_gt, dtype=np.float32), np.array(kps_preds,
    dtype=np.float32)
104
105 def calcRMSError(kps_gt, kps_preds):
106     N = kps_gt.shape[0] * (kps_gt.shape[-1] // 2)
107     error = np.sqrt(np.sum((kps_gt-kps_preds)**2)/N)
108
109     return error
110
111 def show_keypoints(batch_imgs, batch_labels, nrows, ncols,
    predictions=None):
112
113     def draw_keypoints(img, keypoints, col):
114         # print("\n{}".format(len(keypoints)))
115         for i in range(0, len(keypoints)-1, 2):
```

```
116             # print(i)
117             kpx = int(keypoints[i])
118             kpy = int(keypoints[i+1])
119             img = cv.circle(img, center=(kpx,kpy), radius=2, color
    =col, thickness=2)
120 
121         return img
122 
123     fig, axes = plt.subplots(nrows=nrows, ncols=ncols)
124 
125     r = -1
126 
127     for i in range(len(batch_imgs)):
128 
129         img = batch_imgs[i]
130         img = np.reshape(img, newshape=(im_size,im_size))
131         img = np.stack([img,img,img], axis=-1)
132 
133         c = i % ncols
134 
135         if i % ncols == 0:
136             r += 1
137 
138         # draw ground-truth keypoints on image
139         if batch_labels is not None:
140             img = draw_keypoints(img, batch_labels[i], col
    =(0,0,255))
141 
142         # draw predicted keypoints on image
143         if predictions is not None:
144             img = draw_keypoints(img, predictions[i], col
    =(0,255,0))
145 
146         axes[r, c].imshow(img)
147 
148         # one liner to remove *all axes in all subplots*
149     plt.setp(plt.gcf().get_axes(), xticks=[], yticks=[]);
150     plt.show()
```

heatmaps_processing.py

```
import numpy as np
import os
import math
import pandas as pd
import cv2 as cv
import random
import matplotlib.pyplot as plt
from heatmaps_generator import MaskGenerator

import tensorflow
import keras.backend as K
from keras.applications import ResNet50
from keras.models import Model
import keras.utils
from keras.optimizers import Adam
from keras.layers import Input, Conv2D, Flatten, Dense,
    GlobalAveragePooling2D
from keras.callbacks import TensorBoard, ProgbarLogger,
    ModelCheckpoint, ReduceLROnPlateau, EarlyStopping

from unet_model import UNET, trainModel, show_keypoints,
    calcKeypoints

log_dir = 'logs/'
unet_dir = 'unet_logs/'


#def display_heatmaps(batch_images, batch_heatmaps, batch_kpoints,
     nrows, ncols, inc_preds= False, preds=None):
def display_heatmaps(batch_images, batch_heatmaps, nrows, ncols,
    inc_preds= False, preds=None):
  def plot_keypoints(image, keypoints, col):
    for i in range(0, len(keypoints)-1, 2):
      kpoint_x = int(keypoints[i]*image.shape[0])
      kpoint_y = int(keypoints[i+1]*image.shape[1])
      image = cv.circle(image, center=(kpoint_x, kpoint_y), radius
    =2, color=col, thickness=2)
    return image
```

```
33
34    if not inc_preds:
35      nrows -= 1
36
37    figure, axes = plt.subplots(nrows=nrows, ncols=ncols)
38    r=-1
39
40    for i in range(ncols):
41      image = batch_images[i]
42      image = np.reshape(image, newshape=(224,224))
43      image = np.stack([image, image, image], axis=-1)
44
45      #image = plot_keypoints(image, batch_kpoints[i], col=(0,255,0)
      )
46
47      mask = batch_heatmaps[i]
48      mask = np.reshape(mask, newshape=(224,224, 11))
49      mask = np.sum(mask, axis=-1)
50
51      axes[0, i].imshow(image)
52      axes[1, i].imshow(mask)
53
54      if inc_preds:
55        pred_mask = preds[i]
56        pred_mask = np.reshape(pred_mask, newshape=(224, 224, 11))
57        pred_mask = np.sum(pred_mask, axis=-1)
58        axes[2, i].imshow(pred_mask)
59
60    plt.show()
61
62  #Load the training data
63  file_path = "train_images"
64  file = 'Keypoints_Images.csv'
65  df_train = pd.read_csv(file)
66  n_train = df_train['Images'].size
67  df_kpoints = df_train.iloc[:, 0:22]
68
69  indices = []
70
```

```
image_dict = {}
kpoint_dict = {}

for i in range(n_train):
  indices.append(i)
  image_dict[i] = "train"+str(i)+".png"
  kpoint = df_kpoints.iloc[i].values.tolist()
  kpoint_dict[i] = kpoint

#Split the data into train/Val sets
random.shuffle(indices)
end_index = int(0.9*len(indices))
#small_end_index = int(0.995*len(indices)) #  This is just for keypoint-heatmap evaluation to reduce computation time
train_indices = indices[0:end_index]
validation_indices = indices[end_index:len(indices)]
#small_validation_indices = indices[small_end_index:len(indices)] # This is just for keypoint-heatmap evaluation to reduce time

print(f"Number of Training Images: {len(train_indices)}")
print(f"Number of Validation Images: {len(validation_indices)}")
#print(f"Number of Small Validation Images: {len(small_validation_indices)}")

train_data = MaskGenerator(file_path,
               train_indices,
               image_dict,
               kpoint_dict
               )

validation_data = MaskGenerator(file_path,
               validation_indices,
               image_dict,
               kpoint_dict
               )

print(f"Number of Training Batches: {len(train_data)}")
print(f"Number of Validation Batches: {len(validation_data)}")

```

```
#train_images, train_masks, train_keypoints = train_data[0]
train_images, train_masks = train_data[0]

print(f"Training Images Shape: {train_images.shape}")
print(f"Training Heatmaps Shape: {train_masks.shape}")

#display_heatmaps(train_images[0:4], train_masks[0:4],
    train_keypoints[0:4], nrows=3, ncols=4)

unet = UNET(input_shape=(224, 224, 1))
#print(unet.summary())
loss_type = "mse"
unet = trainModel(unet, "unet", loss_type, n_epochs=20, old_lr=1e
    -3, new_lr=1e-4, train_gen=train_data, val_gen=validation_data,
     load_saved_wts=True)
```

keypoint_processing.py

```
import numpy as np
import os
import pandas as pd
import cv2 as cv
import random
import matplotlib.pyplot as plt
from custom_classes import CustomGenerator

import tensorflow
import keras.backend as K
from keras.applications import ResNet50
from keras.models import Model
import keras.utils
from keras.optimizers import Adam
from keras.layers import Input, Conv2D, Flatten, Dense,
    GlobalAveragePooling2D
from keras.callbacks import TensorBoard, ProgbarLogger,
    ModelCheckpoint, ReduceLROnPlateau, EarlyStopping

log_dir = 'logs/'

def display_keypoints(batch_images, batch_kpoints, nrows, ncols,
```

```
    preds=None):
  def plot_keypoints(image, keypoints, col):
    for i in range(0, len(keypoints)-1, 2):
      kpoint_x = int(keypoints[i]*image.shape[0])
      kpoint_y = int(keypoints[i+1]*image.shape[1])
      image = cv.circle(image, center=(kpoint_x, kpoint_y), radius
    =2, color=col, thickness=2)
    return image

  figure, axes = plt.subplots(nrows=nrows, ncols=ncols)

  r=-1

  for i in range(len(batch_images)):
    image = batch_images[i]
    image = np.reshape(image, newshape=(224,224))
    image = np.stack([image, image, image], axis=-1)

    c = i % ncols

    if i % ncols == 0:
      r += 1

    image = plot_keypoints(image, batch_kpoints[i], col=(0,0,255))


    if preds is not None:
      image = plot_keypoints(image, preds[i], col=(255,0,0))

    axes[r, c].imshow(image)

  plt.show()

#Load the training data
file_path = "train_images"
file = 'Keypoints_Images.csv'
df_train = pd.read_csv(file)
n_train = df_train['Images'].size
df_kpoints = df_train.iloc[:, 0:22]
```

```
58
59 indices = []
60
61 image_dict = {}
62 kpoint_dict = {}
63
64 for i in range(n_train):
65   indices.append(i)
66   image_dict[i] = "train"+str(i)+".png"
67   kpoint = df_kpoints.iloc[i].values.tolist()
68   kpoint_dict[i] = kpoint
69
70 #print(image_dict)
71
72 #Split the data into train/Val sets
73 random.shuffle(indices)
74 end_index = int(0.9*len(indices))
75 train_indices = indices[0:end_index]
76 validation_indices = indices[end_index:len(indices)]
77
78 print(f"Number of Training Images: {len(train_indices)}")
79 print(f"Number of Validation Images: {len(validation_indices)}")
80
81 train_data = CustomGenerator(file_path,
82               train_indices,
83               image_dict,
84               kpoint_dict
85               )
86
87 validation_data = CustomGenerator(file_path,
88               validation_indices,
89               image_dict,
90               kpoint_dict
91               )
92
93 print(f"Number of Training Batches: {len(train_data)}")
94 print(f"Number of Validation Batches: {len(validation_data)}")
95
96 train_images, train_keypoints = train_data[0]
```

```
97 #print(f"Training Images Shape: {train_images.shape}")
98 #print(f"Training Keypoints Shape: {train_keypoints.shape}")
99
100 #display_keypoints(train_images[0:12], train_keypoints[0:12],
       nrows=3, ncols=4)
101
102 # Loading and freezing pre-trained model
103 K.set_learning_phase(0)
104 pretrained_model = ResNet50(weights=None, include_top=False,
105
       input_shape=(224, 224, 3))
106 pretrained_model.trainable = False
107
108 input_tensor = Input(shape=(224,224,1) )
109 x_in = Conv2D(3,(3,3),padding='same')(input_tensor)   # x has a
       dimension of (IMG_SIZE,IMG_SIZE,3)
110 pretrained_model = pretrained_model (x_in)
111
112 # Adding new trainable hidden and output layers to the model
113 #K.set_learning_phase(1)
114 x = pretrained_model
115 #x = pretrained_model.output
116 #x = Flatten()(x)
117 x = GlobalAveragePooling2D()(x)
118 #x = Dense(1024, activation="relu")(x)
119 predictions = Dense(22, activation="linear")(x)
120 model_final = Model(inputs=input_tensor, outputs=predictions)
121
122 logging = TensorBoard(log_dir=log_dir)
123 checkpoint = ModelCheckpoint(log_dir + 'ep{epoch:03d}-loss{loss:.3
       f}-val_loss{val_loss:.3f}.h5',
124     monitor='val_loss', save_weights_only=True, save_best_only=
       True, period=3)
125 reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
       patience=3, verbose=1)
126 early_stopping = EarlyStopping(monitor='val_loss', min_delta=0,
       patience=10, verbose=1)
127
128 model_final.compile(loss="mean_squared_error", optimizer=Adam(lr=1
```

```
    e-3))

'''
model_final.summary()

# Training the model (transfer learning)
history = model_final.fit_generator(
    train_data,
    epochs=10,
    initial_epoch=0,
    validation_data=validation_data,
    callbacks=[logging, checkpoint])

model_final.save_weights(log_dir + 'trained_weights_stage_1.h5')

print('Training losses: ', history.history['loss'])
print('Validation losses: ', history.history['val_loss'])
'''

# Unfreeze and continue training, to fine-tune.
# Train longer if the result is not good.
for i in range(len(model_final.layers)):
    model_final.layers[i].trainable = True

model_final.compile(loss="mean_squared_error", optimizer=Adam(lr=1
    e-4)) # recompile to apply the change
print('Unfreeze all of the layers.')

model_final.load_weights('logs/trained_weights_finalTwo.h5')

print('Loaded the roughly trained_weights and beginning training')

#print('Train on {} samples, val on {} samples, with batch size
    {}.'.format(len(train_labels), len(validation_labels),
    batch_size))
final = model_final.fit_generator(
    train_data,
    epochs=50,
    initial_epoch=0,
```

```
164     validation_data=validation_data,
165     callbacks=[logging, checkpoint, reduce_lr])
166
167 model_final.save_weights(log_dir + 'trained_weights_finalThree.h5'
    )
168
169 print('Training losses: ', final.history['loss'])
170 print('Validation losses: ', final.history['val_loss'])
```

D. DPU-based Inference Processing

input_preprocessing.py

```
1  from PIL import Image
2  import numpy as np
3
4
5  def letterbox_image(image, size):
6      '''resize image with unchanged aspect ratio using padding'''
7      iw, ih = image.size
8      w, h = size
9      scale = min(w/iw, h/ih)
10     nw = int(iw*scale)
11     nh = int(ih*scale)
12
13     image = image.resize((nw,nh), Image.BICUBIC)
14     new_image = Image.new('RGB', size, (128,128,128))
15     new_image.paste(image, ((w-nw)//2, (h-nh)//2))
16     return new_image
17
18 #image = Image.open(img_path)
19
20 def preprocessing_fn(image, model_image_size=(416,416)):
21     if model_image_size != (None, None):
22         assert model_image_size[0]%32 == 0, 'Multiples of 32
       required'
23         assert model_image_size[1]%32 == 0, 'Multiples of 32
       required'
24         boxed_image = letterbox_image(image, tuple(reversed(
       model_image_size)))
```

```
    else:
        new_image_size = (image.width - (image.width % 32), image.
    height - (image.height % 32))
        boxed_image = letterbox_image(image, new_image_size)
    image_data = np.array(boxed_image, dtype='float32')
    image_data /= 255.
    return image_data

calib_image_dir = "./images/"
calib_image_list = "./calibration.txt"
calib_batch_size = 1
def calib_input(iter):
  images = []
  line = open(calib_image_list).readlines()
  for index in range(0, calib_batch_size):
    curline = line[iter * calib_batch_size + index]
    image_name = curline.strip()
    image = Image.open(calib_image_dir + image_name)
    image = preprocessing_fn(image)
    images.append(image)
  return {"input_1": images}
```

dnnc_script.sh

```
#!/usr/bin/env bash

net="customyolo"
CPU_ARCH="arm64"
DNNC_MODE="debug"
dnndk_board="Ultra96"
dnndk_dcf="./dcf/Ultra96.dcf"

echo "Compiling Network ${net}"

# Work space directory
work_dir=$(pwd)

# Path of TensorFlow quantization model
model_dir=${work_dir}/quantize_results
# Output directory
```

```
output_dir="dnnc_output"

tf_model=${model_dir}/deploy_model.pb

DNNC=dnnc

# Get DNNDK config info
if [ ! -f /home/kiruki/dnndk/etc/dnndk.conf ]; then
    echo "Error: Cannot find /etc/dnndk.conf"
    exit 1
else
    tmp=$(grep "DNNDK_VERSION=" /home/kiruki/dnndk/etc/dnndk.conf)
    dnndk_version=${tmp#DNNDK_VERSION=}
    dnndk_version=${dnndk_version#v}
    echo "DNNDK      : $dnndk_version"
    echo "Board Name : $dnndk_board"
    echo "DCF file   : $dnndk_dcf"
fi

if [ ! -d "$model_dir" ]; then
    echo "Can not found directory of $model_dir"
    exit 1
fi

[ -d "$output_dir" ] || mkdir "$output_dir"


echo "CPU Arch   : $CPU_ARCH"
echo "DNNC Mode  : $DNNC_MODE"
echo "$(dnnc --version)"
$DNNC   --parser=tensorflow                          \
        --frozen_pb=${tf_model}                      \
        --output_dir=${output_dir}                   \
        --dcf=${dnndk_dcf}                           \
        --mode=${DNNC_MODE}                          \
        --cpu_arch=${CPU_ARCH}                       \
        --net_name=${net}
```

References

[1] Weibo Liu et al. 'A survey of deep neural network architectures and their applications'. In: *Neurocomputing* 234 (2017), pp. 11–26.

[2] Vivek Kothari, Edgar Liberis and Nicholas D Lane. 'The Final Frontier: Deep Learning in Space'. In: *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*. 2020, pp. 45–49.

[3] Karl Weiss, Taghi M Khoshgoftaar and DingDing Wang. 'A survey of transfer learning'. In: *Journal of Big data* 3.1 (2016), p. 9.

[4] Jeremy Fowers et al. 'A configurable cloud-scale DNN processor for real-time AI'. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 1–14.

[5] Norman P Jouppi et al. 'In-datacenter performance analysis of a tensor processing unit'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[6] Nataliia Kussul et al. 'Deep learning classification of land cover and crop types using remote sensing data'. In: *IEEE Geoscience and Remote Sensing Letters* 14.5 (2017), pp. 778–782.

[7] David J Lary et al. 'Machine learning in geosciences and remote sensing'. In: *Geoscience Frontiers* 7.1 (2016), pp. 3–10.

[8] Bradley Denby and Brandon Lucia. 'Orbital edge computing: Machine inference in space'. In: *IEEE Computer Architecture Letters* 18.1 (2019), pp. 59–62.

[9] Jennifer Alvarez and Buddy Walls. 'Constellations, clusters, and communication technology: Expanding small satellite access to space'. In: *2016 IEEE Aerospace Conference*. IEEE. 2016, pp. 1–11.

[10] Inigo Del Portillo, Bruce G Cameron and Edward F Crawley. 'A technical comparison of three low earth orbit satellite constellation systems to provide global broadband'. In: *Acta Astronautica* 159 (2019), pp. 123–135.

[11] Angadh Nanjangud et al. 'Robotics and AI-enabled on-orbit operations with future generation of small satellites'. In: *Proceedings of the IEEE* 106.3 (2018), pp. 429–439.

[12] Daniela Girimonte and Dario Izzo. 'Artificial intelligence for space applications'. In: *Intelligent Computing Everywhere*. Springer, 2007, pp. 235–253.

[13] Jeremy Straub. 'A review of spacecraft AI control systems'. In: *Proc. 15th World Multi-Conference on Systemics, Cybernetics and Informatics*. 2011.

[14] Stephen G Ungar et al. 'Overview of the earth observing one (EO-1) mission'. In: *IEEE Transactions on Geoscience and Remote Sensing* 41.6 (2003), pp. 1149–1159.

[15] John P Grotzinger et al. 'Mars Science Laboratory mission and science investigation'. In: *Space science reviews* 170.1-4 (2012), pp. 5–56.

[16] Max Bajracharya, Mark W Maimone and Daniel Helmick. 'Autonomy for mars rovers: Past, present, and future'. In: *Computer* 41.12 (2008), pp. 44–50.

[17] Cuebong Wong et al. 'Adaptive and intelligent navigation of autonomous planetary rovers—A survey'. In: *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE. 2017, pp. 237–244.

[18] Raymond Francis et al. 'AEGIS autonomous targeting for the Curiosity rover's ChemCam instrument'. In: *2015 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*. IEEE. 2015, pp. 1–5.

[19] David A Whelan et al. 'Darpa orbital express program: effecting a revolution in space-based systems'. In: *Small Payloads in Space*. Vol. 4136. International Society for Optics and Photonics. 2000, pp. 48–56.

[20] Marc D Rayman et al. 'Results from the deep space 1 technology validation mission'. In: *Acta Astronautica* 47.2-9 (2000), pp. 475–487.

[21] Nicola Muscettola et al. 'Remote agent: To boldly go where no AI system has gone before'. In: *Artificial intelligence* 103.1-2 (1998), pp. 5–47.

[22] Jay Wyatt et al. 'Flight validation of on-demand operations: The deep space one beacon monitor operations experiment'. In: *Artificial Intelligence, Robotics and Automation in Space*. Vol. 440. 1999, p. 357.

[23] Elizabeth M Middleton et al. 'The earth observing one (EO-1) satellite mission: Over a decade in space'. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 6.2 (2013), pp. 243–256.

[24] Kiri L Wagstaff et al. 'Cloud filtering and novelty detection using onboard machine learning for the EO-1 spacecraft'. In: *Proc. IJCAI Workshop AI in the Oceans and Space*. 2017.

[25] Steve Chien et al. 'Onboard science processing concepts for the HyspIRI mission'. In: *IEEE Intelligent Systems* 24.6 (2009), pp. 12–19.

[26] Steve Chien et al. 'Onboard autonomy on the intelligent payload experiment (IPEX) CubeSat mission: a pathfinder for the proposed HyspIRI mission intelligent payload module'. In: *Proc 12th International Symposium in Artificial Intelligence, Robotics and Automation in Space, Montreal, Canada*. 2014.

[27] Yohei Iwasaki et al. 'Development and Initial On-orbit Performance of Multi-Functional Attitude Sensor using Image Recognition'. In: *33rd Annual AIAA/USU Conference on Samll Satellites, Utah, USA*. 2019.

[28] Kirk Woellert et al. 'Cubesats: Cost-effective science and technology platforms for emerging and developing nations'. In: *Advances in Space Research* 47.4 (2011), pp. 663–684.

[29] Veronica L Foreman, Afreen Siddiqi and Olivier De Weck. 'Large satellite constellation orbital debris impacts: Case studies of oneweb and spacex proposals'. In: *AIAA SPACE and Astronautics Forum and Exposition*. 2017, p. 5200.

[30] Antoine Petit, Eric Marchand and Keyvan Kanani. 'Tracking complex targets for space rendezvous and debris removal applications'. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 4483–4488.

[31] Vladimir Aslanov and Vadim Yudintsev. 'Dynamics of large space debris removal using tethered space tug'. In: *Acta Astronautica* 91 (2013), pp. 149–156.

[32] Minghe Shan, Jian Guo and Eberhard Gill. 'Review and comparison of active space debris capturing and removal methods'. In: *Progress in Aerospace Sciences* 80 (2016), pp. 18–32.

[33] Angel Flores-Abad et al. 'A review of space robotics technologies for on-orbit servicing'. In: *Progress in Aerospace Sciences* 68 (2014), pp. 1–26.

[34] European Space Agency. *Pose Estimation Challenge*. (Accessed on 03 August 2019).

[35] Yifan Lu et al. 'A review of the space environment effects on spacecraft in different orbits'. In: *IEEE Access* 7 (2019), pp. 93473–93488.

[36] L.M Martines S. *Analysis of LEO Radiation Environment and Its Effects on Spacecraft's Critical Electronic Devices*. 2011.

[37] NASA. *The Radiation Environment*.

[38] Isai Fajardo et al. 'Design, Implementation, and Operation of a Small Satellite Mission to Explore the Space Weather Effects in LEO'. In: *Aerospace* 6.10 (2019), p. 108.

[39] A. D. George and C. M. Wilson. 'Onboard Processing With Hybrid and Reconfigurable Computing on Small Satellites'. In: *Proceedings of the IEEE* 106.3 (2018), pp. 458–470. DOI: 10.1109/JPROC.2018.2802438.

[40] George Lentaris et al. 'High-performance embedded computing in space: Evaluation of platforms for vision-based navigation'. In: *Journal of Aerospace Information Systems* 15.4 (2018), pp. 178–192.

[41] Xilinx. *RT Kintex UltraScale FPGAs for Ultra High Throughput and High Bandwidth Applications*. 2020.

[42] ESA. *Phi-Sat 1 Mission*. (Accessed on 02 October 2020).

[43] Kaiyuan Guo et al. '[DL] A survey of FPGA-based neural network inference accelerators'. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.1 (2019), pp. 1–26.

[44] Gianmarco Dinelli et al. 'MEM-OPT: A Scheduling and Data Re-Use System to Optimize On-Chip Memory Usage for CNNs On-Board FPGAs'. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 10.3 (2020), pp. 335–347.

[45] Xuechao Wei et al. 'Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs'. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[46] Xiaocong Lian et al. 'High-performance fpga-based cnn accelerator with block-floating-point arithmetic'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8 (2019), pp. 1874–1885.

[47] Matthieu Courbariaux et al. 'Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1'. In: *arXiv preprint arXiv:1602.02830* (2016).

[48] Mohammad Rastegari et al. 'Xnor-net: Imagenet classification using binary convolutional neural networks'. In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[49] Yaman Umuroglu et al. 'Finn: A framework for fast, scalable binarized neural network inference'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 65–74.

[50] Michaela Blott et al. 'FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks'. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–23.

[51] Jiantao Qiu et al. 'Going deeper with embedded fpga platform for convolutional neural network'. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 26–35.

[52] Kaiyuan Guo et al. 'Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2017), pp. 35–47.

[53] Kaiyuan Guo et al. 'From model to FPGA: Software-hardware co-design for efficient neural network acceleration'. In: *2016 IEEE Hot Chips 28 Symposium (HCS)*. IEEE. 2016, pp. 1–27.

[54] Xilinx. *DNNDK User Guide*. (Accessed on 26 September 2019).

[55] Xilinx. *DPU Product Guide*. (Accessed on 09 August 2020).

[56] Jiang Zhu et al. 'An Efficient Task Assignment Framework to Accelerate DPU-Based Convolutional Neural Network Inference on FPGAs'. In: *IEEE Access* 8 (2020), pp. 83224–83237.

[57] Deepak Kumar. *Verilog Tutorial*. (Accessed on 21 November 2017).

[58] Wiley. *Verilog HDL Design*. (Accessed on 26 March 2020).

[59] Doulos. *VHDL Guide*. (Accessed on 06 September 2019).

[60] David Kriesel. *A Brief Introduction to Neural Networks*. 2005.

[61] James Diebel. 'Representing attitude: Euler angles, unit quaternions, and rotation vectors'. In: *Matrix* 58.15-16 (2006), pp. 1–35.

[62] Yaguang Yang. 'Spacecraft attitude determination and control: Quaternion based method'. In: *Annual Reviews in Control* 36.2 (2012), pp. 198–219.

[63] R Mukundan and KR Ramakrishnan. 'A quaternion solution to the pose determination problem for rendezvous and docking simulations'. In: *Mathematics and computers in simulation* 39.1-2 (1995), pp. 143–153.

[64] Roberto Opromolla et al. 'A review of cooperative and uncooperative spacecraft pose determination techniques for close-proximity operations'. In: *Progress in Aerospace Sciences* 93 (2017), pp. 53–72.

[65] Simone D'Amico, Mathias Benn and John L Jørgensen. 'Pose estimation of an uncooperative spacecraft from actual space imagery'. In: *International Journal of Space Science and Engineering 5* 2.2 (2014), pp. 171–189.

[66] Daichi Hirano, Hiroki Kato and Tatsuhiko Saito. 'Deep Learning based Pose Estimation in Space'. In: *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*. 2018.

[67] Sumant Sharma. 'Pose estimation of uncooperative spacecraft using monocular vision and deep learning'. PhD thesis. Stanford University, 2019.

[68] Thaweerath Phisannupawong et al. 'Vision-Based Spacecraft Pose Estimation via a Deep Convolutional Neural Network for Noncooperative Docking Operations'. In: *Aerospace* 7.9 (2020), p. 126.

[69] Aiden Nibali et al. 'Numerical coordinate regression with convolutional neural networks'. In: *arXiv preprint arXiv:1801.07372* (2018).

[70] Alexander Toshev and Christian Szegedy. 'Deeppose: Human pose estimation via deep neural networks'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 1653–1660.

[71] Jonathan J Tompson et al. 'Joint training of a convolutional network and a graphical model for human pose estimation'. In: *Advances in neural information processing systems*. 2014, pp. 1799–1807.

[72] Alejandro Newell, Kaiyu Yang and Jia Deng. 'Stacked hourglass networks for human pose estimation'. In: *European conference on computer vision*. Springer. 2016, pp. 483–499.

[73] Yu Chen et al. 'Adversarial posenet: A structure-aware convolutional network for human pose estimation'. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 1212–1221.

[74] Xiao Sun et al. 'Integral human pose regression'. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 529–545.

[75] Joseph Redmon and Ali Farhadi. 'Yolov3: An incremental improvement'. In: *arXiv preprint arXiv:1804.02767* (2018).

[76] Hesham Amin, K Memy Curtis and Barrie R Hayes-Gill. 'Piecewise linear approximation applied to nonlinear function of a neural network'. In: *IEE Proceedings-Circuits, Devices and Systems* 144.6 (1997), pp. 313–317.

[77] MT Tommiska. 'Efficient digital implementation of the sigmoid function for reprogrammable logic'. In: *IEE Proceedings-Computers and Digital Techniques* 150.6 (2003), pp. 403–411.

[78] Jordan L Holi and J-N Hwang. 'Finite precision error analysis of neural network hardware implementations'. In: *IEEE Transactions on Computers* 42.3 (1993), pp. 281–290.