1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Ricardo Manuel Carriço Barreto

IoT Edge Computing Neural Networks on Reconfigurable Logic

Dissertation submitted to the Department of Electrical and Computer Engineering of the Faculty of Science and Technology of the University of Coimbra in partial fulfilment of the requirements for the Degree of Master of Science

September 2019

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Ricardo Manuel Carriço Barreto

IoT Edge Computing Neural Networks on Reconfigurable Logic

Thesis submitted to the

University of Coimbra for the degree of
Master in Electrical and Computer Engineering

Supervisors:
Prof. Dr. Jorge Nuno de Almeida e Sousa Almada Lobo
Prof. Dr. Paulo Jorge Carvalho Menezes

Coimbra, 2019

This work was developed in collaboration with:

University of Coimbra

Department of Electrical and Computer Engineering

Institute of Systems and Robotics

Dedication

Dedico este trabalho primeiro aos meus pais, que sempre me apoiaram para fazer o que eu mais gosto, mas especilmente durante os últimos anos, e me deram todas as ferramentas necessárias para terminar o meu percurso académico com sucesso. Aos meus irmãos, pois a vida é melhor com irmãos, também agradeço os momentos bem passados em família, e todas as vezes que me chateavam, pois nem tinha piada se os irmãos não se chateassem uns aos outros. A uma pessoa muito especial, à minha namorada agradeço por toda a paciência e o apoio incondicional nestes anos de faculdade, me segurou e motivou por muitas vezes, o que permitiu chegar aqui. Agradeço também a todo o pessoal do lab por toda a ajuda que me deram no decorrer deste trabalho. Aos meus amigos, por todos os momentos bem passados, em especial aos Brunos e Gustavo por todos os momentos, desde a constante inter-ajuda até às pausas para o café, pelas noites passadas no DEEC para acabar os projetos, mas sempre com muita diversão.

Ricardo Barreto

Acknowledgements

Quero agradecer especialmente aos meus supervisores, Prof. Dr. Jorge Lobo e Prof. Dr. Paulo Menezes da Universidade de Coimbra por todas as conversas, sugestões e orientações durante a execução deste projeto.

Agradeço também ao Instituto de Sistemas e Robotica (ISR) pela cedência do espaço e de todo o material necessário para o desenvolvimento deste trabalho.

Abstract

In recent years we have seen the emergence of AI in wider application areas.However, in the IoT ecosystem there is the tendency to use cloud computing to store and process the vast amounts of information generated by these devices, due to the limited local resources. This dissertation proposes the implementation of smart IoT devices able to provide specific information from raw data produced from some sensor, e.g. a camera or microphone, instead of the raw data itself. The focus will be embedded image processing using Convolutional Neuronal Networks (CNN). This approach is clearly distinct from the current trends in IoT devices that use cloud computing to process the collected data. We intend a twist on the established paradigm and pursue an edge computing approach. Since we are targeting small and simple devices, we need some low power solution for the CNN computation. SoC devices have gained popularity due to their heterogeneity. In our work we use a system that combines an Hard Processor System (HPS) unit in conjunction with FPGA, while maintaining low power consumption, taking advantage of FPGA to achieve high performance. HADDOC2 was used as a tool that converts a CNN model to VHDL code to be synthesized to FPGA, while in HPS there is a system that manages the entire process using IoT communication protocols to send the processed information. A system with a CNN implemented in the FPGA is obtained using the HPS linux program to manage all the subsystems and sending the processed data through the MQTT protocol for IoT.

Resumo

Nos últimos anos, temos visto a expanção da inteligência artificial em diferentes áreas e dispositivos. No entanto, no ecossistema IoT, temos uma tendência constante a usar a computação na nuvem para armazenar e processar as vastas quantidades de dados geradas por estes dispositivos, devido aos recursos locais limitados. Esta dissertação propõe a implementação de dispositivos IoT inteligentes capazes de fornecer informações específicas a partir de dados produzidos a partir de algum sensor, por exemplo uma câmara ou microfone, em vez dos próprios dados brutos. O foco será o processamento de imagens usando CNNs. Essa abordagem é claramente distinta das tendências atuais em dispositivos IoT que usam computação na nuvem para processar os dados produzidos. Pretendemos uma viragem no paradigma estabelecido e procuramos uma abordagem de *edge computing*. Como o foco serão dispositivos pequenos e simples, precisamos de uma solução de baixa potência para o cálculo da CNN. Os dispositivos SoC ganharam popularidade devido à sua heterogeneidade. Este trabalho usará um sistema que combina uma unidade de processamento ARM em conjunto com a FPGA, mantendo baixo consumo energético e explorando a FPGA para obter um alto desempenho. O programa HADDOC2 foi usado para converter um modelo de uma CNN para código VHDL a ser sintetizado para a FPGA, enquanto no ARM existe um sistema que gere todos os processos usando pontes de comunicação com a FPGA e protocolos de comunicação IoT para enviar as informações processadas. No fim é obtido um sistema com uma CNN implementada na FPGA o usando o HPS como gestor de todo o processo e que se comunica com o exterior através do MQTT.

Acronyms

AI	Artificial Intelligence.
ARM	Advanced RISC Machine.
ASIC	Application Specific Integrated Circuits.
CNN	Convolutional Neural Networks.
CPU	Central Processing Unit.
DHM	Direct Hardware Mapping.
DMA	Direct Memory Access.
DNN	Deep Neural Networks.
FPGA	Field Programmable Gate Array.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
HPS	Hard Processor System.
IC	Integrated Circuit.
IoT	Internet of Things.
MQTT	Message Queuing Telemetry Transport.
NN	Neural Networks.

OCM	On-Chip Memory.
OS	Operating System.
SDRAM	Synchronous dynamic random-access memory.
SoC	System On Chip.
VHDL	VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.

List of Figures

Contents

Chapter 1

Introduction

1.1 Motivation

The search to make physical devices intelligent has always existed. Even when the first computer was developed, it was intended to draw conclusions and get results faster than humans can. Embedding Artificial Intelligence (AI) capabilities on devices can be an interesting solution for improving their sensing capabilities (human-like) or performing in-situation pattern recognition for producing e.g. semantic interpretations of the raw data, instead of the raw data itself. Among the AI techniques available Neural Networks (NN) and in particular Convolutional Neural Networks (CNN) have demonstrated very interesting capabilities in performing pattern recognition in images. These networks are currently running on computers with large processing capacity, which are bulky and consume huge amounts of energy, which is a problem for remote and resource-constrained solutions. When creating an Internet of Things (IoT) device it needs to fulfil some requirements such as low energy, connectivity, reduced size (Patel & Patel, 2016), and developing custom circuits is often required. The development may make use of Application Specific Integrated Circuits (ASIC), which are custom made chips for specific applications, having a great optimisation, energy efficient and high performance, but a fixed design and high production costs. This constrain limits the modularity of the system and implies a thorough and flawless development. On the other hand, Field

Programmable Gate Array (FPGA)s are devices with the ability to run dedicated circuits that can be reconfigured at any time during development or later in the field. This maintains the modularity without having to spend a lot of money on production and testing. In addition to being a reconfigurable dedicated circuit they can have lower power consumption and power saving than Central Processing Unit (CPU) and Graphics Processing Unit (GPU).

The constant growth of IoT leads to the massive production of data that needs to be processed. The cloud computing data centers have been growing in recent years and are being used on a global scale. But IoT devices are growing at a breakneck speed, which will lead to cloud service outages and existing bandwidth may no longer be sufficient for the desired demand, which may lead to data processing problems. So the paradigm needs to change and one of the solutions is local processing, so called edge computing.

1.2 Related Work

In order to contextualise this work with the state of the art some works related to this theme are presented and will be discussed.

Analysing the growth of IoT in various areas, the work by (Shi, Cao, Zhang, Li, & Xu, 2016) shows how beneficial it would be to use edge computing, thus changing the current paradigm in data processing, and reducing bandwidth and increasing efficiency.

The work (Pena, Rodriguez-Andina, & Manic, 2017) shows the IoT and the internet revolution with the entry of these devices into everyday life, is stated that the impact of IoT will be as big as the emergence of the internet. One of the biggest problems with IoT is that the same concept is applied in many different areas. Thus different architectures have to be developed for different needs. While in some areas data security and privacy is crucial, in others the continuity of the service is crucial. IoT systems are very heterogeneous, which, in turn, greatly complicates control and management tasks. The emergence of System On Chip (SoC) FPGAs combining the reconfigurable circuits with the Advanced RISC Machine (ARM) processor increases the versatility, flexibility, performance, security and scalability of the IoT projects.

Due to the low processing and bandwidth constraints of IoT devices, new approaches have been emerging to perform data processing locally. Thinking of a new strategy to realise the processing of CNN in the IoT devices, the work of (Du et al., 2017) process a CNN in streaming, making use of a dedicated processor for IoT devices as processing unit. The CNN data flow has been optimised to decrease the number of times it is accessed, thereby increasing CNN's computing efficiency. The results were favourable, being a great strategy for IoT devices, similar to the CNN implementation of HADDOC2 (Abdelouahab, Pelcat, Sérot, Bourrasset, & Berry, 2017) in FPGA.

Some works have explored the implementation of NN and Deep Neural Networks (DNN) in FPGAs. In (Wei et al., 2017) a systolic array architecture is used to process CNN in FPGAs. A high capacity FPGA, an Intel Aria 10 board, was used. This system takes advantage of the FPGA by parallelizing the data processing, achieving up to 1.2 Tops for 8-16 bit fixed point data.

The work (Nurvitadhi et al., 2017) directly compares DNN processing on FPGAs and GPUs. This makes use of Intel's latest Aria 10 and Stratix 10 FPGAs and compares with the latest Nvidia titan X GPU. It took advantage of the DNN's sparsity and all unnecessary operations have been removed. This achieved very satisfactory results for the FPGAs by beating the GPU in the performance tests.

To combat one of the major problems of FPGAs such as the difficult development with FPGAs, complexity and a long time-to-market led to the creation of tools that would facilitate developers in their projects. This gives rise to tools that automate the Hardware Description Language (HDL) code creation, in this case from CNN models. Some frameworks to convert CNN models to hardware design language have been and continue to be developed. CONDOR (Raspa, Bacis, Natale, & D. Santambrogio, 2018) is an automated framework to implement CNN on FPGA. This framework is under development and the objective is to map CNN computation in hardware taking advantage of the parallelism of the CNN. Another framework is fpgaConvNet that in addition to custom mapping a CNN in the FPGA, uses the specifications of the board to create a more optimised code to the platform (Venieris & Bouganis, 2017). However, neither of the above are open-source, and since is intended an open software to improve and customise, they were not used in this work. HADDOC2 is a framework that has the ability to convert CNN caffe models to VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (VHDL) code (Abdelouahab et al., 2017). Using Direct Hardware Mapping (DHM), HADDOC2 is taking maximum advantage of network parallelism by implementing it without any pipeline. So it takes full advantage of the FPGA. It is also not limited by the platform and the source code is open-source. This creates an advantage to developers to modify and improve the current HADDOC2 work.

1.3 Objectives

A key objective of this work is to validate the use of reconfigurable logic as a means to have simple IoT devices capable processing data and only relaying curated data to the internet. Opposed to cloud computing, our focus is on edge computing. By exploring the flexibility of SoC systems that include hard processors and an FPGA, the aim is to have a full operational pipeline (figure 1.1).

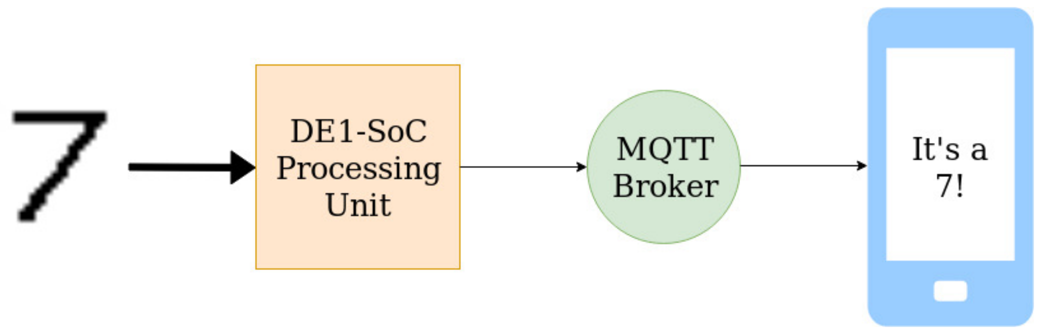


Figure 1.1: Envisioned processing pipeline, with an IoT device capturing an image, performing local computations, and sending to the internet processed data.

1.4 Key Contributions

This work stands out for the use of SoC technology that is increasingly used to build heterogeneous systems, taking advantage of a low power ARM processor and accelerating a CNN in the FPGA, thus obtaining a system that can be implemented in the world of IoT devices.

- An IoT device using the MQTT protocol sending the processed data to a broker (server);
- It was built the necessary HPS structure to the data management and interrupt handler;
- In the FPGA it was built a system that manages the memory, interrupts and the HADDOC2 CNN;
- The HADDOC2 framework contained several architectural issues in the FPGA design. The entire structure has been revised and the necessary modules redesigned. Only the conversion of CNN weights remained uncorrected;
- At the end, the CNN classification is not the expected, but all the necessary tools to ramp up this project are assembled;
- With all this tools is achieved a full pipeline implemented in a SoC device.

1.5 Thesis Structure

This thesis is structured by presenting a first chapter with the general introduction, motivation and related work. Chapter 2 describes the various tools used in this project. The system architecture is described in chapter 3 showing several diagrams for ease of understanding. Chapter 4 goes deeper into architecture by showing the fundamental developments made in this work. In chapter 5 there is a discussion showing the obtained results. This work ends in Chapter 6 with a conclusion of the work and present the future work.

Chapter 2

Neural Networks, Internet of Things and Reconfigurable Logic

2.1 Artificial Neural Networks

The name artificial intelligence began to be discussed in the middle of the twentieth century (McCarthy, Minsky, Rochester, & Shannon, 2006) following a research proposal by Dartmouth College. However, due to the processing limitations of the time only in the 21st century did Artificial Intelligence (AI) regain importance, becoming part of our daily lives.

Artificial intelligence has gained prestige in several areas for its good results in the perception of patterns collected from large databases. As a subset of AI, Neural Networks (NN) appeared as inspiration of the biological network of brains, learning how to perform tasks through a training process that consists on feeding the network with large amounts of data, so that they learn the patterns, without having to be manually programmed.

The NN are based on connections and nodes, called artificial neurons (figure 2.1). Each connection, like a synapse, passes information from one neuron to another.

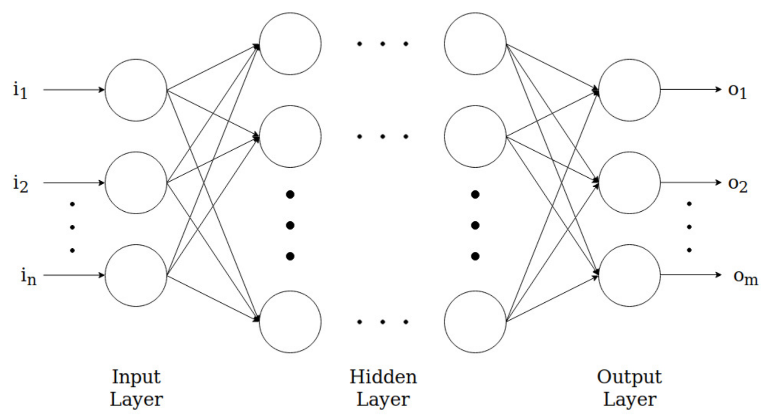


Figure 2.1: Structure of neural network.

As can be seen in figure 2.2 each neuron performs a summation of the multiplication of all inputs with the corresponding weight plus a bias (2.1a). Then the result goes through an activation function (2.1b), that is usually a non-linear function.

$$a_i = \sum_{k=1}^{r} x_k * w_{i,k} + b_i \tag{2.1a}$$

$$y_i = f(a_i) \tag{2.1b}$$

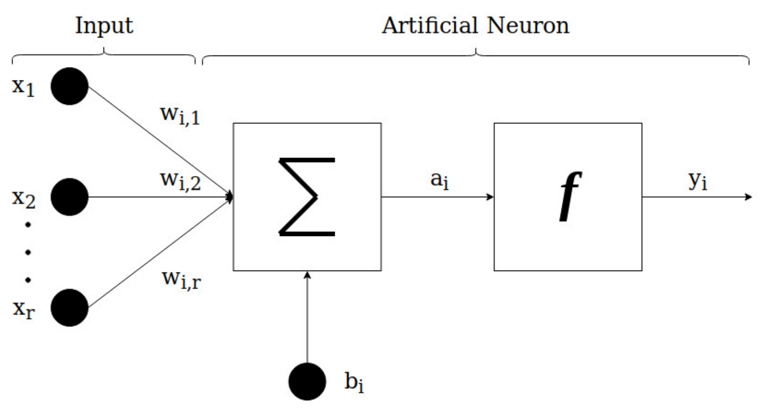


Figure 2.2: Neuron of neural network.

2.1.1 Convolutional Neural Networks

Artificial neural networks with several convolution layers are called Convolutional Neural Networks (CNN), which are a part of Deep Neural Networks (DNN). As these networks have several layers, and each layer contains several convulsions, it is possible to have an accurate extraction of image characteristics, finalising with the fully connected layer that calculates a probability of each output, and thus obtaining a result.

In figure 2.3 a small CNN network used in this work is shown. This network has multiple layers, and each convolution layer has several different filter convolutions with the input image to extract the maximum number of features. At the end of each image convolution there is an activation function (not represented in the image), usually a nonlinear function as in the neuronal networks like a sigmoid, tanh (hyperbolic tangent) or ReLU (Rectified Linear Unit) function. The pooling layer is a filter used to remove redundant points, leaving only the maximum value of that region. The CNN ends with the fully connected layer which assigns a final rating from the features extracted in convolution.

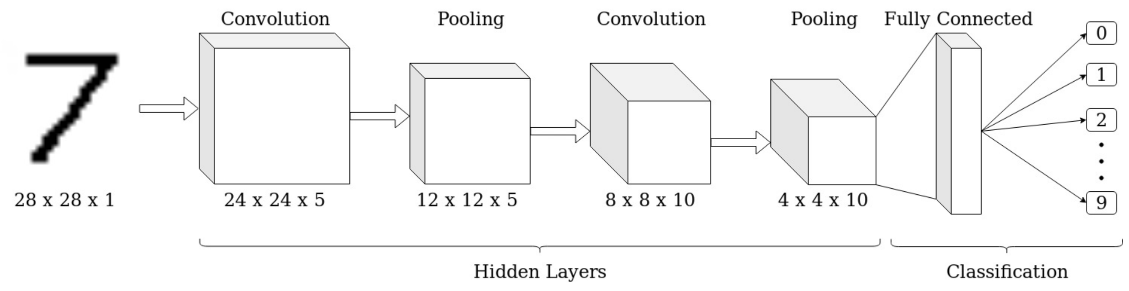


Figure 2.3: Convolution neural network.

2.2 Internet of Things

The system goal is to have high processing power while maintaining a small size and energy efficient. Together with the heterogeneity of the system, it has all the valences to become an IoT device. Thus, for communication with the outside world, a communication protocol related to IoT was chosen. Some data protocols for IoT communication are the CoAP (Constrained Application Protocol), DDS (Data Distribution Service) and Message Queuing Telemetry Transport (MQTT). The CoAP bases on the UDP protocol does not ensure data delivery. The DDS is similar to the Message Queuing Telemetry Transport (MQTT), based on the TCP protocol, being reliable and having decentralised nodes. The MQTT have a centralised node, the broker (server), so it suit in this work ((Chen & Kunz, 2016)).

The MQTT is a TCP based protocol published by IBM and then open-sourced for messaging applications. It works in a publish-subscribe format, where clients can publish a message to a specific topic in the broker (server), and this message will be published to the clients that have subscribed to the same topic (figure 2.4). This enables a lightweight implementation, with security and reliability. Another characteristics that suit this project is the low bandwidth consumption by the MQTT protocol and the hide community support.

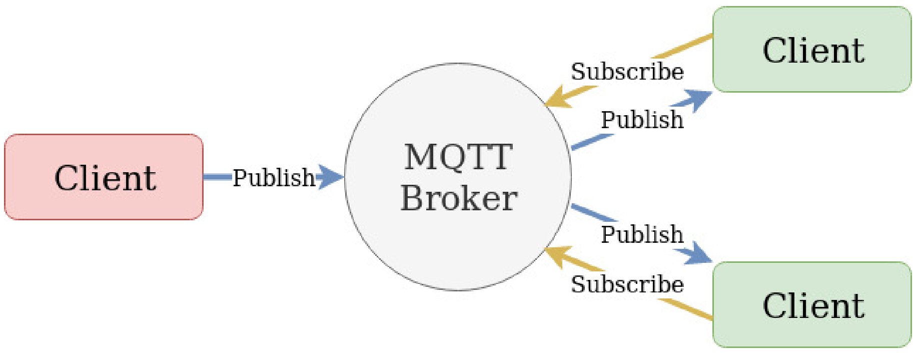


Figure 2.4: MQTT diagram.

2.3 NN Frameworks

2.3.1 HADDOC2

One of the biggest difficulties in the world of reconfigurable systems is the slow and painful development.To face this problem, various tools are beginning to appear, helping and facilitating the developing of dedicated circuits. The HADDOC2 is a tool that automatically converts the CNN model to VHDL code for the FPGA, making its use practical and flexible. Also the framework is opensource, making possible to develop and improve the current code.

The CNN are widely used in pattern perception in images and object recognition. It also have a unique structure to become a great candidate for using FPGAs. It has various convolutions per layer, containing filters that multiply with the same image, making possible the use of full power of FPGA by maintaining a synchronous system and parallelizing its entire structure, obtaining the shortest path and a high throughput.

As shown in figure 2.5, the framework uses Caffe CNN models as input and generates the corresponding VHDL code to be synthetized and then deployed in the FPGA.

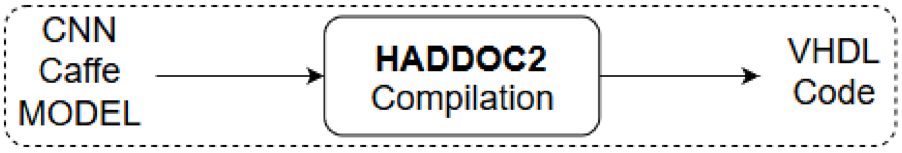


Figure 2.5: HADDOC2 operation.

The HADDOC2 framework creates a full design of the network in the FPGA, called Direct Hardware Mapping (DHM). The compilation process maps to hardware code a complete CNN, which means that all neurons in a layer are mapped on the device to take advantage of inter-neuron parallelism, and each convolution is mapped separately and, finally, each multiplier is instantiated separately. This method occupies a considerable space of FPGA hardware.

Since the code generated by HADDOC2 does not depend on the used FPGA, an "infinite" circuit is created. This means that if the CNN is too big, in the end the circuit may not fit in the FPGA. The only parameter that can be changed in the HADDOC2 compilation is the size of the bitwidth used in the network. As it was used a low-end educational board, FPGA's usable space is very limited, so in this project it was used a small LeNet character detection CNN.

To decrease the space used in the FPGA, the HADDOC2 implements some methods to optimise the final code. The number of multiplications is reduced by detecting redundant operations, multiplications by 0 removed, by 1 replaced by a single connection and by a power of 2 are transformed into shift registers.

A major issue encountered while using this tool was the miss-synchronization of the FPGA code that performs the CNN, that results in poorly performed operations. There were also compilation errors in the code synthesis. Thus, we proceeded to study, correct and develop this tool to achieve the desired processing as will be explained in detail in the following chapter.

2.3.2 Caffe Deep Learning Framework

To convert the CNN model to VHDL code the HADDOC has to receive as input a specific model. The framework was developed to work only with the models created by the Caffe (Jia et al., 2014). Caffe is a deep learning framework, one of the first to turn up, which was developed by Berkeley AI Research (BAIR), who made it open-source so it continues to grow with the help of a wide community.

2.4 Reconfigurable Logic Systems

2.4.1 DE1-SoC Terasic Board

For this project, as a processing platform for code development and testing, the Terasic DE1-SoC Development Kit board.This is a entry level board widely used for educational purposes. It includes the Altera Cyclone V SoC which includes a dual-core Cortex-A9 ARM processor (Hard Processor System (HPS)) with a re-configurable FPGA that has about 85K logic elements and 4450 Kbits of On-Chip Memory (OCM), 1 Gbyte of Synchronous dynamic random-access memory (SDRAM) and communication bridges to easily operate both sides. Figure 2.6 shows the peripherals connected to the Cyclone V. This diagram shows the different types of memory coupled to FPGA and HPS, as well as the various buttons, switches and LEDs that are connected to the board which facilitates the development and debugging of the system.

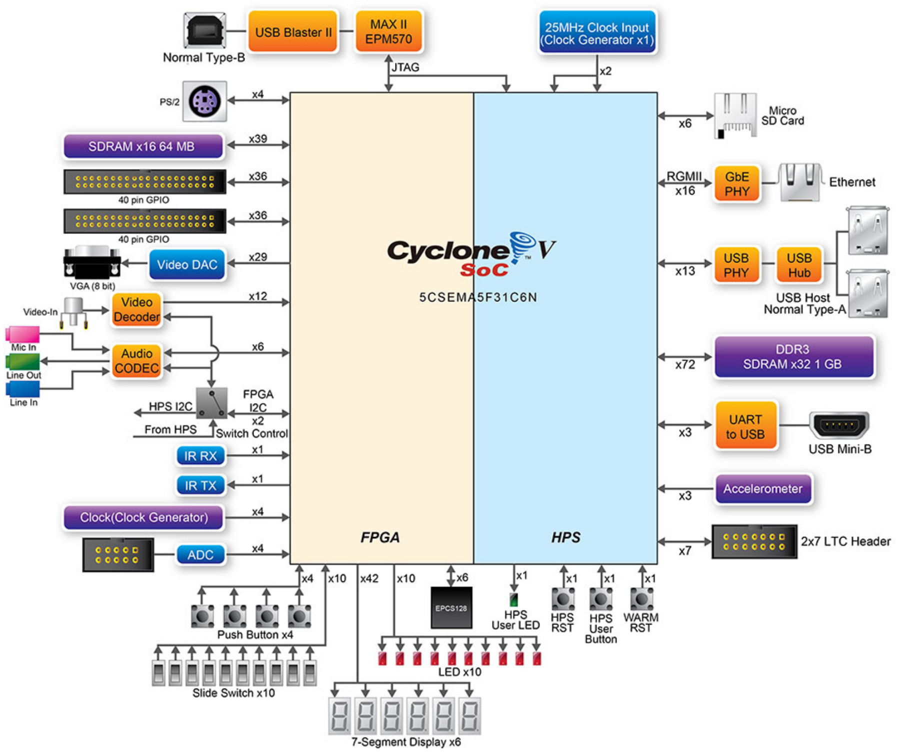


Figure 2.6: CycloneV diagram. (Source: (Terasic, 2019))

2.4.2 FPGA Development Software

The FPGA code developed requires dedicated software. For this board is used the Quartus programmable logic device design software produced by Intel. Inside this there is a SOPC builder (System on a Programmable Chip Builder) called Qsys, that automatically generates interconnect logic of complex dedicated blocks, like HPS-FPGA system, dedicated processors, memory management (SDRAM, OCM), interfaces, protocols and peripherals. This tool is referred later, separating the code developed for FPGA, from the mounted system using Qsys.

2.4.3 HPS-FPGA bridges

The DE1-SoC board contemplate a cyclone V Integrated Circuit (IC). This chip has both the HPS processor and the FPGA.

To easily send information from the HPS to the FPGA the board has some mapped bridges. As seen in figure 2.7 there are 3 bridges connecting the HPS and the FPGA. Also there is a bridge connecting the FPGA to the SDRAM controller subsystem. This bridge allows the FPGA to have full access of the SDRAM.

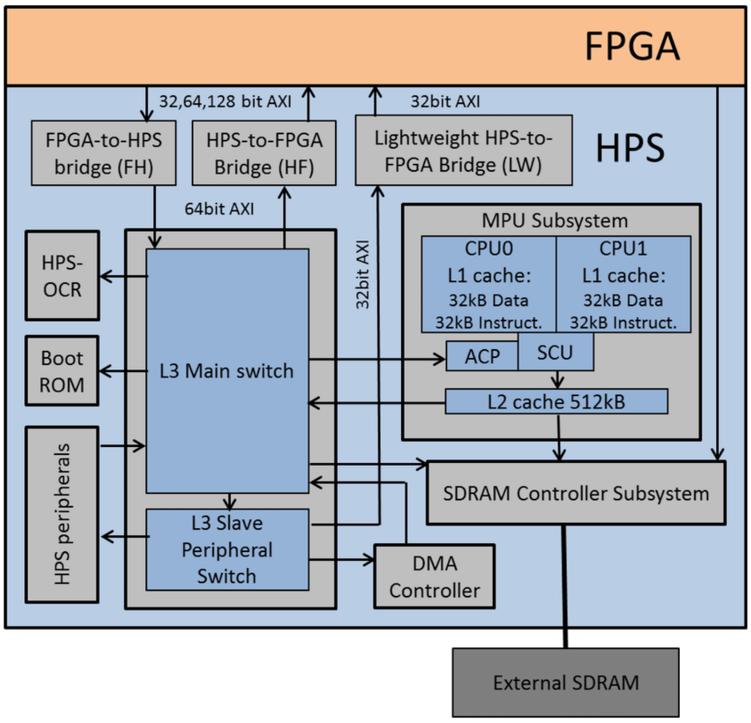


Figure 2.7: Cyclone V diagram. (Source: (Molanes, Rodriguez-Andina, & Farina, 2017))

Chapter 3

Proposed System Architecture

3.1 Overview

This project intend to make use of the reconfigurable ability of FPGAs to accelerate the calculations of a CNN, maintaining a low power device usable in an IoT field. For this, several components were added in order to get a pipeline from the input of the image to be processed in the system, until the output of the data. This way the high level architecture showed in figure 3.1 was reached. This image shows as input the image source, then the use of a development board DE1-SoC as control system, memory management, interrupt handler and CNN processing. At the end the data is sent to the MQTT broker that can be reached by various devices.

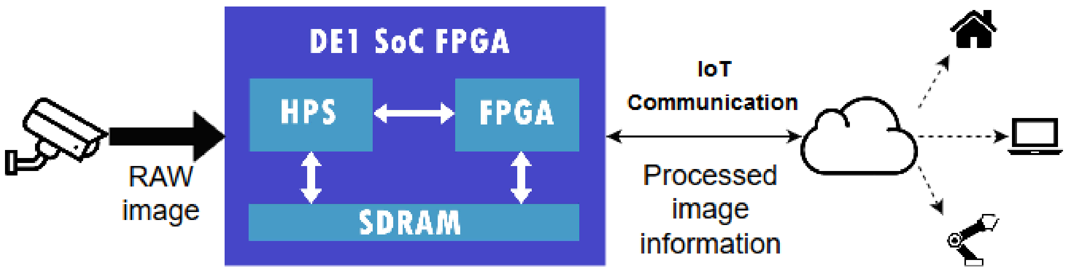


Figure 3.1: High level architecture.

3.2 System Operation

To run this project, it is mandatory to have a main program that controls the process flow and manages all data. This way, it was developed a program in C language that will manage every system in DE1-SoC. Figure 3.2 shows the main C program receiving the image, and communicating with memory and FPGA. The program execution steps are as follows:

1. Receive a image as input;
2. Put the image in memory (SDRAM);
3. Activate the CNN processing on FPGA;
4. Receive an interrupt from FPGA representing the end of CNN processing;
5. Sends the data to the broker via MQTT;

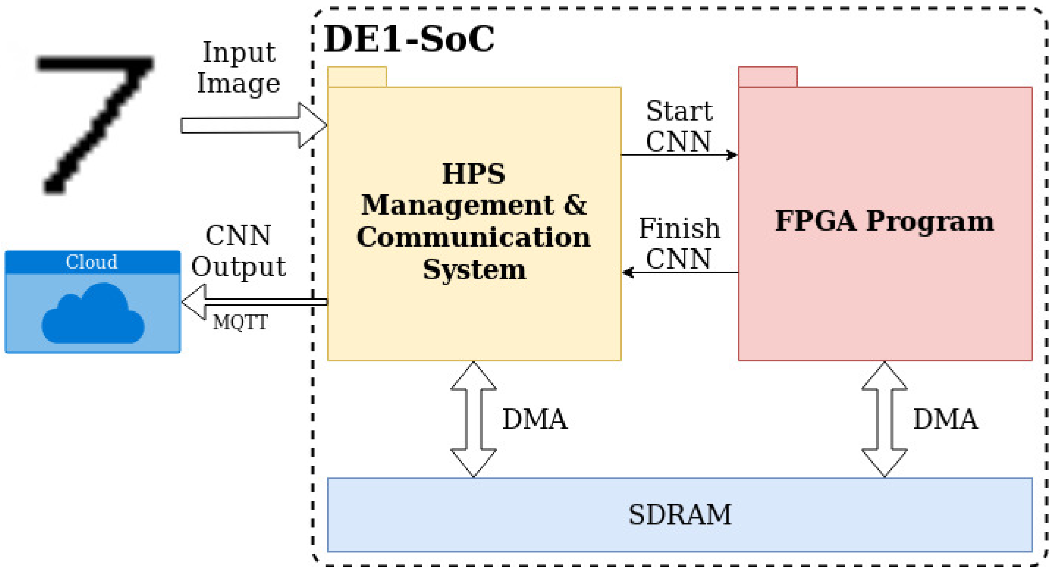


Figure 3.2: HPS and FPGA program in DE1-SoC and its communication

3.3 HPS Management & Communication System

This program runs on the Linux in the HPS. It is responsible to manage all data and keep track on every process. After the image reading, this image is then sent to a specific address of the SDRAM, a shared memory region between the HPS and the FPGA, so the FPGA can then do the processing. Also it manages the interrupt to and from the FPGA using communication bridges. The figure 3.2 shows all the communications that the HPS Management System has.

3.4 FPGA Management System

The HADDOC2 system implemented on the FPGA does not work standalone. It is necessary to create a control system that delivers the data to be processed to the CNN and receives the data on the output. In FPGA the system is divided in two. The modules designed using the Qsys tool and the modules developed in HDL code.

In this way as shown in figure 3.3 a dedicated system was built that reads an image from OCM, sends the image to the CNN, and in the end stores the CNN values in another OCM. An approach to directly read from the SDRAM to the FPGA could be tested, but would involve an increased research and development effort. Since the FPGA contains free memory banks (OCM), these were used to store and read information from global memory SDRAM. In addition, the access to the OCM is simple and intuitive. This system is supported by the *HPS Management & Communication system*, which controls the FPGA memory access.

In Qsys tool 7 modules were used. In figure 3.3 only 5 modules are presented, the other 2 refer to signals sent from HPS to FPGA using the bridges to activate the system, and from FPGA to HPS informing the end of processing. The *HPS-FPGA System* block is a module developed for the CycloneV chip that implements the communication bridges between HPS and FPGA, accessing SDRAM, and allowing access to various board peripherals, interrupt systems, reset and clocks. Both DMAs are controlled from the HPS, through the *HPS-FPGA System.* The data is collected from the SDRAM and placed in the *On-Chip Memory 1*, that is read by the *Read Image.* Then the CNN processing starts and the output is written in the *On-Chip Memory 2* and the hps controller *DMA 2* puts the information in SDRAM.

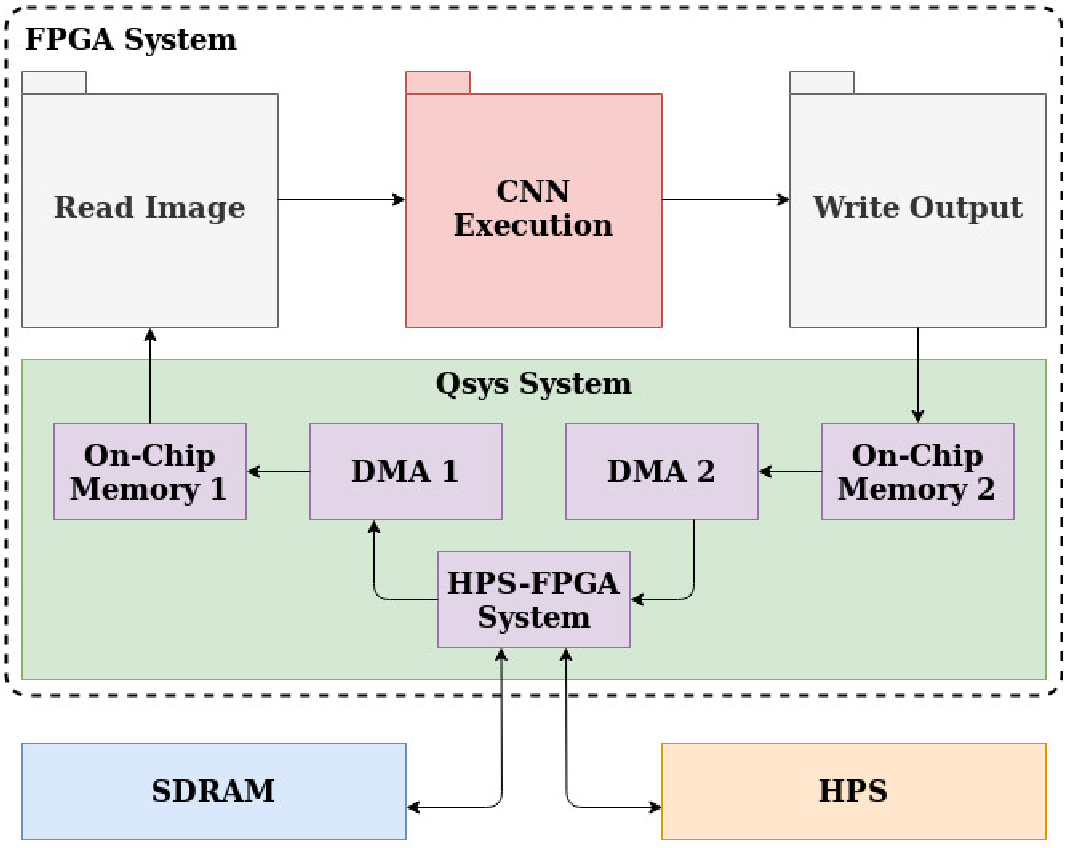


Figure 3.3: FPGA System

3.5 HADDOC2 Architecture

HADDOC2 framework was developed with the intention of making the process of developing dedicated systems using FPGAs simpler and more practical. For this, the framework converts convolutional neural network models to VHDL code, which can then be synthesized for any FPGA device. The HADDOC2 is a *Python* program that generates VHDL code with the parameters of the network. It begins by reading the CNN Caffe model and then generates the corresponding VHDL files. The only parameter that can be changed in the program is the bitwidth size of the network. This way, the network is not dedicated to a specific platform, so the code generated is for a hypothetical infinite space FPGA. This way, the network is not dedicated to a specific platform, so the generated code is for a "infinite" FPGA.

A CNN is constituted by convolutions. These convolutions are masks that contain weights. These weights, the size of convolutions and the fully connected layer is what defines a CNN. In a Caffe model, all weights are distributed between -1 and 1, that is, decimal numbers. FPGAs have some difficulty in dealing with decimal numbers, and when it comes to a large problem, the use of decimal numbers is avoided. Thus HADDOC2 begins by performing a conversion of the positive and negative decimal numbers by a scaling factor. The equation 3.1 receive N as parameter, corresponding to the bitwidth size defined by the user. After multiplying all the numbers with this scale factor, these are converted to binary representation, from 0 to 2N with sign, 2's complement.

$$scale\ factor = 2^{N-1} - 1 \tag{3.1}$$

The figure 3.4 shows the HADDOC2 files used in the FPGA project. The *cnn_process*, *bitwidth* and *params* are the files generated by the HADDOC2 *Python* program. These files have the necessary variables and network parameters. All the other files use these parameters, so the synthesis process instantiate the necessary convolutions (*DotProduct_j* and *TanhLayer_j*) in each *ConvLayer_i*. The *cnn_process* file that is created by the HADDOC2, has the network structure, connecting the *ConvLayer_i* to the *PoolLayer_k* and instantiating the necessary *ConvLayer_i* and *PoolLayer_k*.

So this framework has all the necessary features implemented for a simple CNN. It has the convolution layer, it accepts the tanh activation function, and does the pooling of the image. It also has VHDL custom data types created to easily manage the data in arrays and matrix.

One major problem that was detected is that despite it has all necessary layers for a simple CNN implemented, it has timing problems in *ConvLayer_i* and *PoolLayer_k* functions, and compilations errors. This means that the convolutions are not performed correctly, the multiplications are done with the wrong masks, the pooling is poorly performed, and the propagation of errors through the network makes its use impossible. So the converted CNN could never work properly. In the next chapter, the development and patches made to HADDOC2 will be discussed.

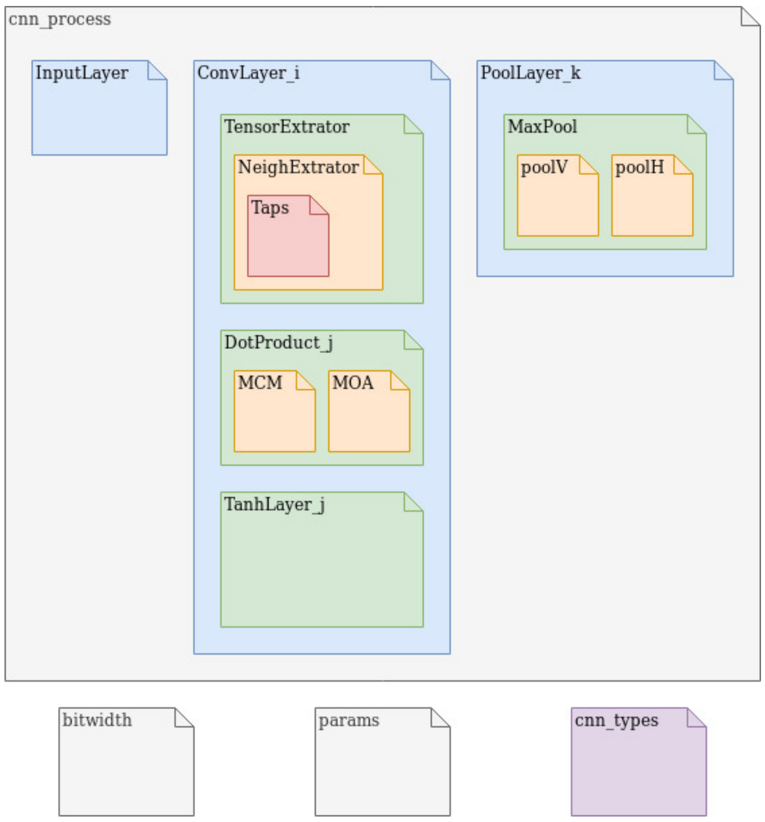


Figure 3.4: HADDOC2 file structure.

Chapter 4

Implementation of Proposed System

After the overview of the general architecture, this chapter addresses implementation details and the steps taken to have an operational system.

4.1 Memory and Bridges Management

As discussed in the previous chapter, the *HPS Management & Communication System* does all the management of the system, controlling the interrupts to and from the FPGA, send the data using the MQTT protocol to the broker and manage the data memory.

4.1.1 Memory Management

The Linux system performs all the management of physical memory. Normally it is not possible to access physical memory directly, but as this is a modified version of the linux kernel, the security that blocked user access to physical memory is disabled. Physical memory is divided into page frames. Virtual memory is organised into pages. The size of the page will match the size of the page frame in physical memory. In this system, each page frame occupies 4096 bytes. When allocating space in memory, linux creates a virtual address that will map to a physical memory zone. If the allocation is less than the page frame then the addresses will be consecutive in both virtual and physical memory. If the allocation is larger than the page frame, the information will be consecutive in virtual memory, but in physical memory it will be filling page frames, but scattered throughout the memory (figure 4.1). This is a problem when HPS and FPGA have the same memory. Since the goal is to give the initial address to the FPGA and from there, knowing the allocated size, it deals with the readings and writings. With the information scattered in memory, the FPGA would have to skip memory addresses. Thus this approach was discarded.

Since this is a development system, it is easy to change system properties. Thus in the boot menu (U-Boot) the memory specifications have been modified to limit its use by linux by 50%. This way there is 50% of memory that can be used and managed by the user to best suit the project needs, and linux will not interfere.

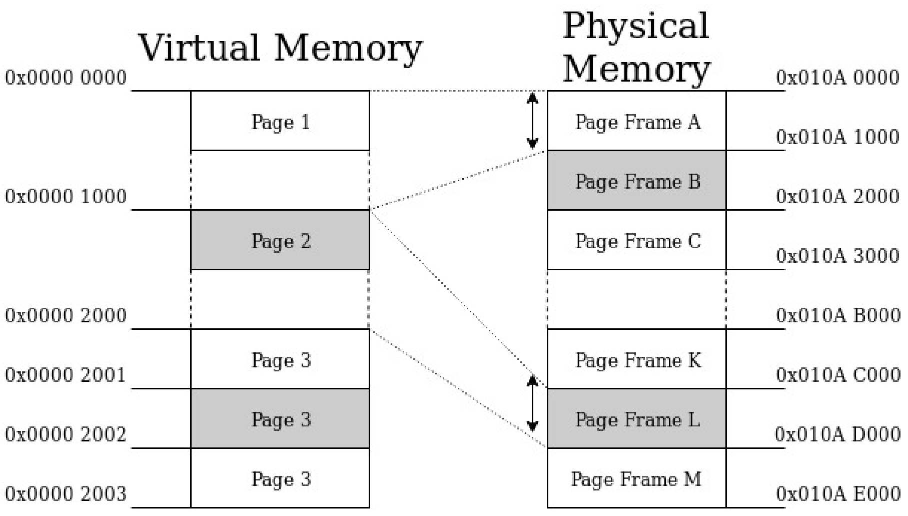


Figure 4.1: Memory mapping.

4.2 Shared Memory

The free space in the SDRAM is used as shared memory between the HPS and the FPGA. The DMA 1 and 2 are Direct Memory Access (DMA) controllers used to copy information from memory spaces. This DMA controllers are created in the

FPGA space, but controlled by the *HPS Management & Communication System*. This system starts by receiving the input image, and place it in the shared memory. After that the DMA 1 is configured with the right addresses, and then activated. It copies the desired content from the SDRAM to the On-Chip Memory 1. After the CNN processing in the FPGA, again, the *HPS Management & Communication System* configures the DMA 2 to copy the information from the On-Chip Memory 2 to the SDRAM. This way, in the end the linux program has access to all information created by the FPGA, accessing it from the SDRAM.

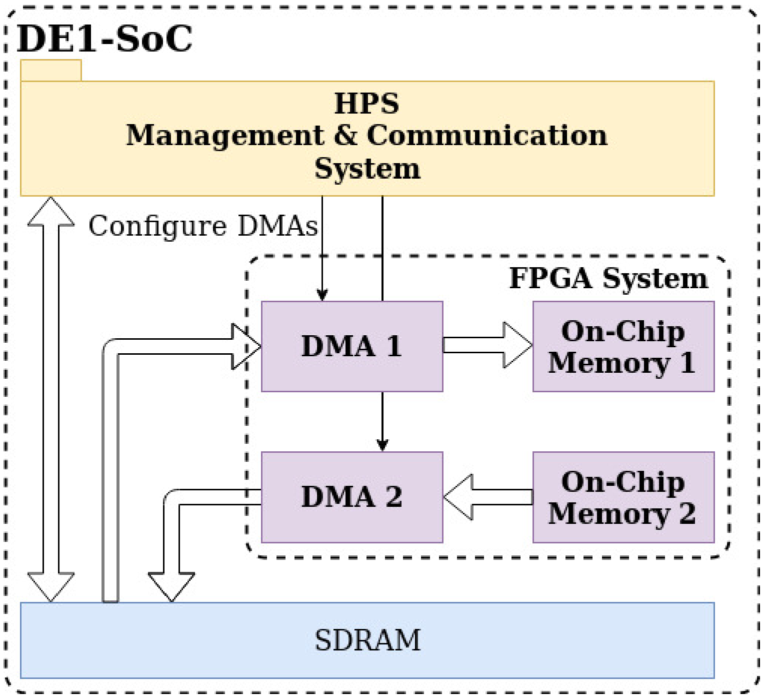


Figure 4.2: Shared memory and memory flow diagram.

4.3 Communication Bridges

A SoC system has many features to take advantage from many different applications. To meet the needs of various types of projects, features have been added that facilitate the development with these boards. The communication bridges are one of these features. This implementation makes it easier to transfer information between HPS and FPGA. It also allows direct connection of these bridges to FPGA I/O devices, memory managers and others.

There are three bridges connecting the HPS and the FPGA as shown in figure 4.3. The first is the HPS-to-FPGA bridge which has 3 different bit-width configurations, 32, 64 and 128 bit. The second is the lightweight HPS-to-FPGA bridge which has only a 32 bits configuration, becoming the most used bridge for it's simple configuration. Both addresses to access these bridges are presented in table 4.1. The third is a FPGA-to-HPS bridge that is used to access information in the HPS side, more common to access memory components.

Bridge	Start Address	End Address
HPS-to-FPGA	0xC000 0000	0xFBFF FFFF
Lightweight HPS-to-FPGA	0xFF20 0000	0xFF3F FFFF

Table 4.1: Bridges start addresses.

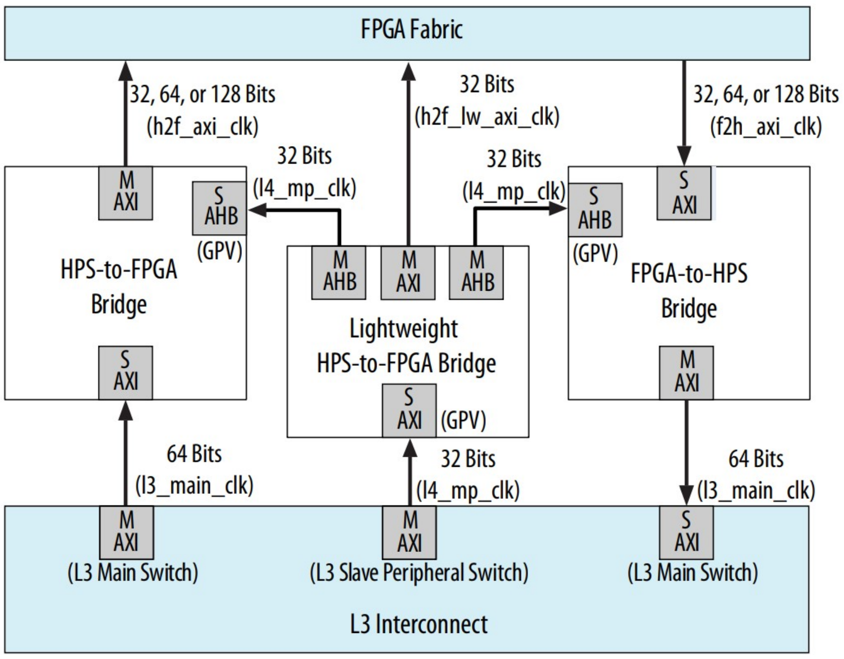


Figure 4.3: DE1-SoC bridges diagram. (Source: (Altera, 2018, p. 612))

The project uses the lightweight HPS-to-FPGA bridge to configure the DMA controllers and manage the interruptions to and from the FPGA.

To communicate with the SDRAM, there is a specific bridge for this purpose, the FPGA-to-SDRAM. This bridge is shown in figure 4.4, and also gives the information that there are up to 6 masters that can connect to the SDRAM. This way, it was created two FPGA-to-SDRAM bridges, one for read and other for writing.

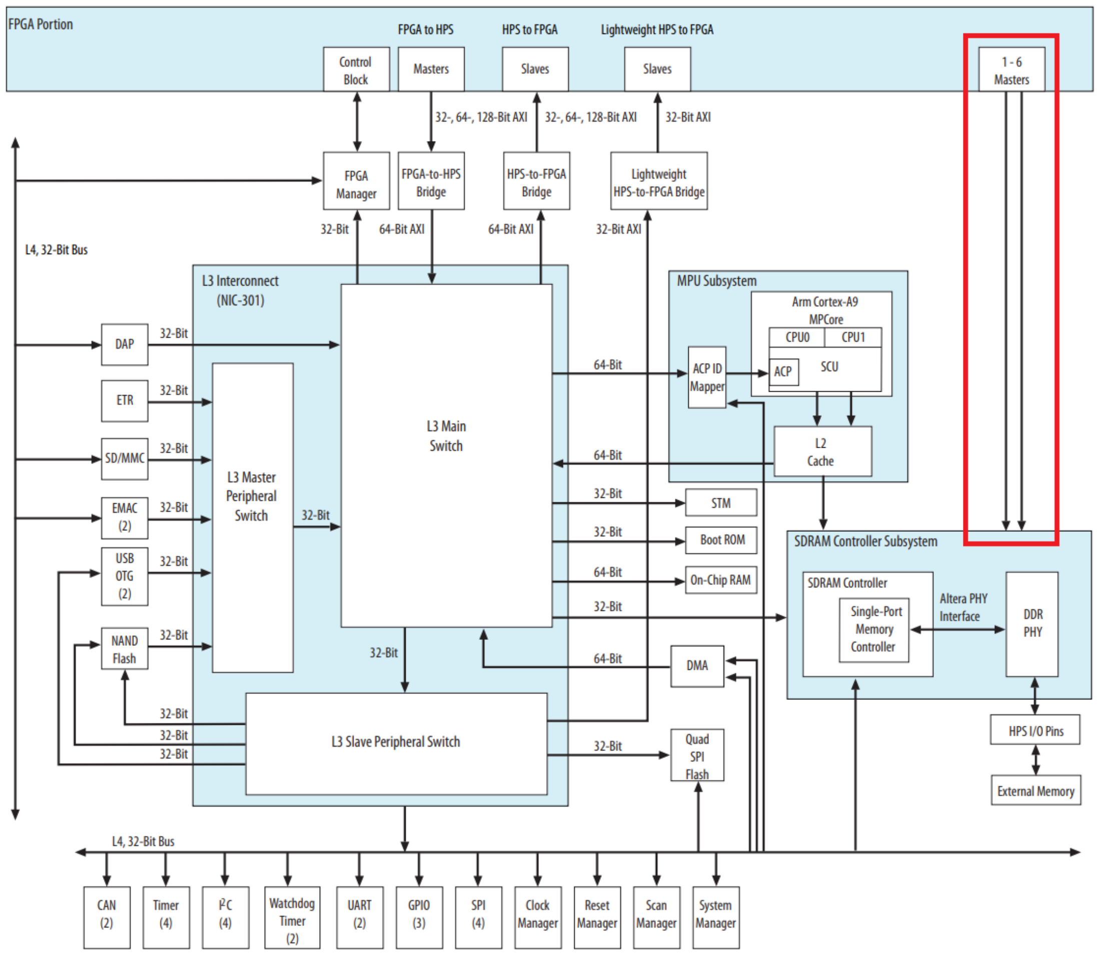


Figure 4.4: DE1-SoC Cyclone V block diagram (Adapted from: (Altera, 2018, p. 43))

4.4 Board Configuration

This project makes use of the DE1-SoC board. This being a development board it has multiple combinations for its functioning. From the back side it has dip-switches as show in figure 4.5 that allows to change the board configuration mode. The two modes used in this project is the stand-alone mode only to program the FPGA and the U-Boot, with image stored in the SD card a Linux with frame buffer distribution. Both modes correspond to the MSEL[4:0] configuration 10010 and 00000 respectively.

A modified version of ubuntu linux was used in this project. This version already has implemented the device driver to communicate with the FPGA, and has enabled the most common bridges for communication with the FPGA.

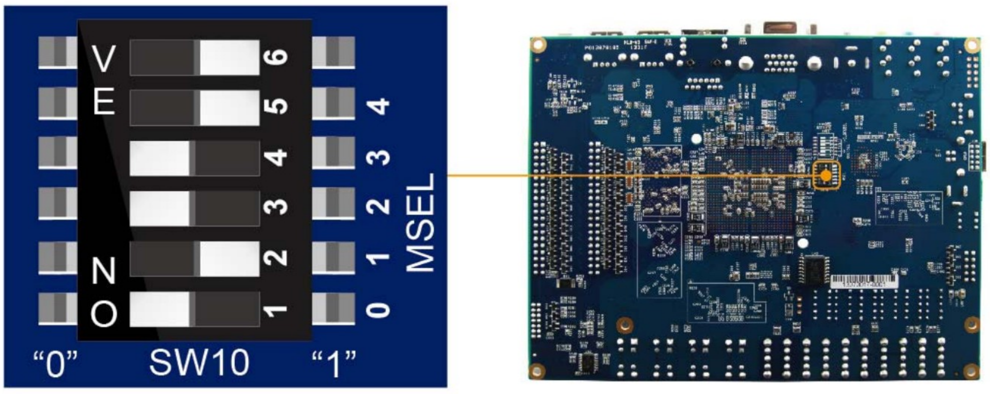


Figure 4.5: DE1-SoC dip switches. (Source: (Terasic, 2015, p. 13))

An older version of ubuntu with more documentation for this board was tried, but the communication with the FPGA is not activated, and the software support has already ended. The Caffe framework and the MQTT protocol also no longer worked in this version.

4.4.1 Boot Stages

The board initialisation process goes through various stages. To make the changes in the memory used by the linux Operating System (OS), and change the bridges configuration, a study to the board boot stages was made. In image 4.6 is shown the boot flow. Following is a description of each step.

1. **Reset** - The CPU 1 on board exits from the reset state and starts running code at the reset exception address. This exception address is usually mapped to the *Boot ROM*.

2. **Boot ROM** - Is responsible to determine the boot source (has mentioned in the previous chapter) and set up the clock manager.

3. **Preloader** - initializes the SDRAM interfaces and configure the I/O pins. Initialising the SDRAM alows the *Preloader* to load the next stage of the bootsoftware, usually a open source boot loader, like U-Boot (universal boot-loader).

4. **Boot Loader** - Loads boot configuration and the OS.

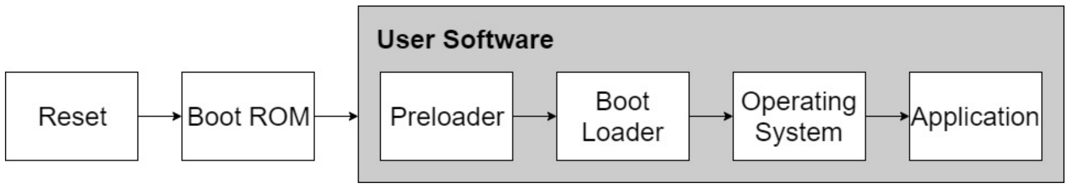


Figure 4.6: DE1-SoC boot stages

This last stage is very important as it is here that the user can change certain initial settings by entering in a interactive command-line session. Here it is possible to change the memory used by the OS, set MAC addresses, enable and disable the communication bridges between the HPS and the FPGA, and other configuration settings.

4.5 HADDOC2

The HADDOC2 tool was originally intended to do all CNN conversion of the VHDL code. However, this was not the case, and after some study it was concluded that several improvements would have to be made to correct the problems of timing between the various modules and compilation errors. Also the documentations of the HADDOC2 does not go into detail in the configuration of the modules. So, after this, an intense reverse engineering journey began to understand all the layers created in HADDOC2 construction.

As shown in the previous chapter, the HADDOC2 creates 3 files when it runs. The compilation error came from here, because the bitwidth must be respected for all parameters. However, the network bias values were not respecting this bitwidth value, and it was higher sometimes. This leads to compilation errors. The solution was to limit the maximum and minimum value of the bias to the value of the scale factor (equation 3.1).

Then the HADDOC2 inputs and outputs were analysed. Originally it has 6 inputs, *clock*, *reset*, *enable*, *in_data*, *in_dv* and *in_fv* as can be seen in figure 4.7. The *clock* signal used in this project is the 50MHz clock. The *reset* input makes a reset of the whole system. The *enable* activates the system. The *in_data* receives a bus width, with the size predefined in the HADDOC2 file creation. Each input data corresponds to a pixel of the image, in stream mode. The *in_dv* is a input that communicates to the HADDOC2 system that is receiving an input data. The *in_fv* is used to communicate the processing of a new image (for run-time image processing).

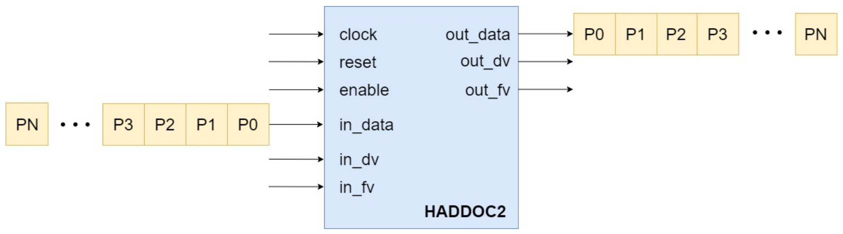


Figure 4.7: HADDOC2 inputs and outputs.

Each time the *in_fv* input was zero, it reset the system to receive a new image. Allied to the interpretation of RTL (register-transfer level) of the circuit, it was concluded that the *in_fv* input was the same as activating the *reset* input, it would put the system in the initial state, ready to receive an image. At the same time, it was realised that the critical path of the circuit would be reduced. Removing this input, also reduces the complexity of the system, reducing the number of multiplexers. To verify the activated inputs, the VHDL code makes use of if conditions. These 1 bit if conditions in design space are represented with multiplexers. The image 4.8 shows what happens in the design space. When the program has a cascade of if conditions, the corresponding RTL circuit creates a cascade of multiplexers. This increases the complexity and the critical path of the system.

The VHDL code has a cascade of if conditions because of the else condition when *in_fv* becomes 0. Removing this condition, it was also possible to join both *enable* and *in_dv* conditions as shown in image 4.9. This way, not only removed a layer of multiplexers, it also removed 2 layers of multiplexers, because the inputs *enable* and *in_dv* were verified using a and logic gate. Almost all HADDOC2 modules performed input checking through if conditions. Thus in these modules, the code has been changed.

The HADDOC2 has 2 major structures, the Convolution layer and the

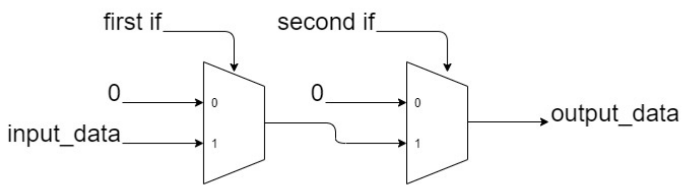
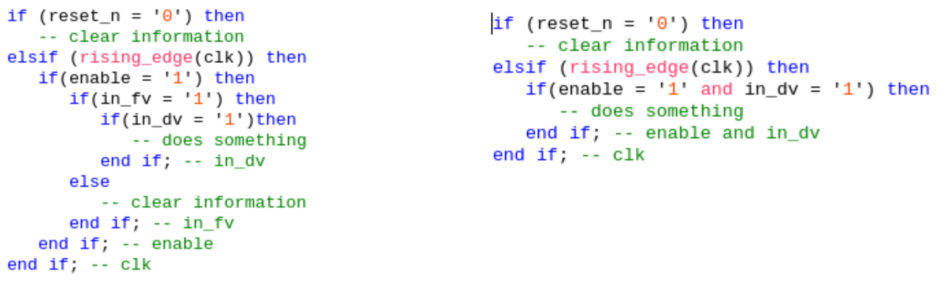


Figure 4.8: If condition in block diagram using multiplexers.



```
if (reset_n = '0') then
   -- clear information
elsif (rising_edge(clk)) then
   if(enable = '1') then
      if(in_fv = '1') then
         if(in_dv = '1')then
            -- does something
         end if; -- in_dv
      else
         -- clear information
      end if; -- in_fv
   end if; -- enable
end if; -- clk
```

(a) Long if example.

```
if (reset_n = '0') then
   -- clear information
elsif (rising_edge(clk)) then
   if(enable = '1' and in_dv = '1') then
      -- does something
   end if; -- enable and in_dv
end if; -- clk
```

(b) Short if example.

Figure 4.9: Different if conditions examples.

Pooling layer. From here, the debugging of the HADDOC2 was divided in two parts, the Convolution layer and the Pooling layer.

4.5.1 Convolution Layer

In order to optimise the number of operations performed by the FPGA and to take advantage of all its parallelism for the execution of convolutions, first there is a process of extracting the necessary values to perform the respective convolutions. Thus, there is a layer "TensorExtrator" that realises the operation represented in image 4.10. As the HADDOC2 receives the pixel stream from the image in each clock cycle, these pixels are stored in a vector with size equal to the equation 4.1. This way the vector is filled with that size. At each clock cycle it is possible to extract the necessary values from the image to convolve these values with a mask. Since CNN can have several convolutions, the image values are divided by several convolution modules, where each will convolve. In this way we use the minimum of space keeping the parallelism in the operations.

The FPGA is a dedicated circuit, it allows to perform several operations in the same clock cycle, and parallelize the multiple convolutions. Thus, each kernel convolution with the image is performed in a clock cycle, multiplying all values. In the next clock cycle, all values are summed using the binary adder tree. This keeps the flow of data and does not have to store the received values again, saving space on the FPGA, and speeding up the CNN process. At the end of the convolution, the data is passed by a activation function, a tanh non linear function.

$$Tensor\ Extrator\ Array = Image\ Width * (Kernel\ Height - 1) + Kernel\ Width \tag{4.1}$$

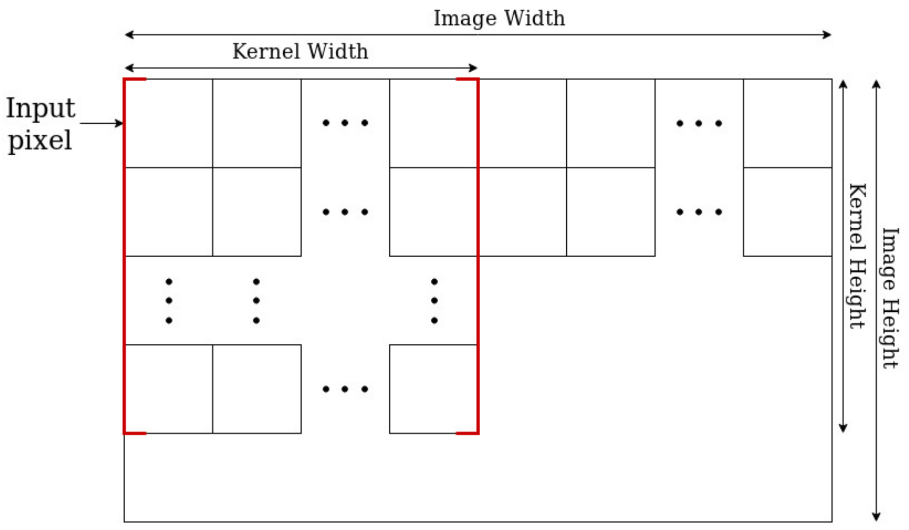


Figure 4.10: Input array for Convolutions.

4.5.1.1 Changes to Convolution Layer

Initially the Convolution Layer has to store the input data. This was done by using a structure called *taps*. This structure would save a line of the image, so the program would create as many *taps*, as the kernel width size. In the figure 4.11 the kernel width number is represented by the letter i. As can be seen, each *taps* passes the following structure the oldest pixel value, ie at the end the structure *taps_i* will have the corresponding oldest line. By passing the value from one structure to another, one value is passed from one register to another. This will create a delay between values in various taps.

This complex structure of independent lines had a lot of issues, first by having a 1 clock cycle delay between all of them (so as more *taps* it had, more delay

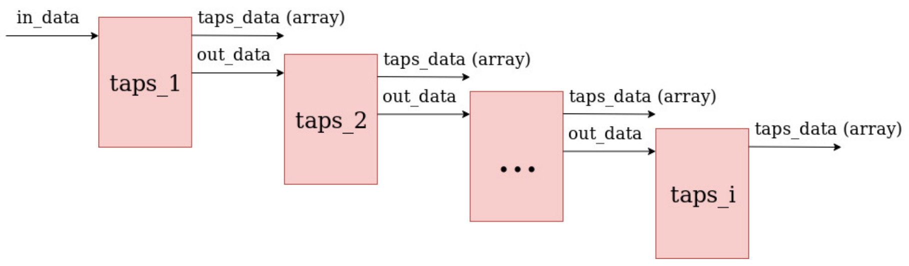


Figure 4.11: HADDOC2 original taps.

the program had), and each one, had the same coding, so it was repeating the circuit in the FPGA, which led to the use of more logical elements. To solve this a single *taps* where created, to store all the necessary values represented in image 4.10, and remove the delay.

The *taps* structure only stores the values not managing the data flow. To control the data flow and activate the *in_dv* output for the next stage, it is needed a controller that counts the number of values received. As represented in the figure 4.10 to start the first convolution with the kernel, it has to receive the number of values represented in equation 4.1. This controller was redesigned and tested leading to perfect results.

4.5.2 Pooling Layer

The HADDOC2 using a sequential process, has to manage the input data so that it uses the fewer possible clock cycles in the FPGA. This way, the pooling layer receives a stream of data corresponding to the filtered image in the convolution layer. Each clock cycle the poolV layer receives the data corresponding to a pixel from the convolution layer, and stores this data in a buffer. The size of the buffer is represented by the equation 4.2.

$$Kernel\ Height - 1 * Image\ Width \tag{4.2}$$

The image 4.12 represents the array of the pooling module containing the data from the convolution layer. This module does the maximum of *Pool1* and *Pool2*

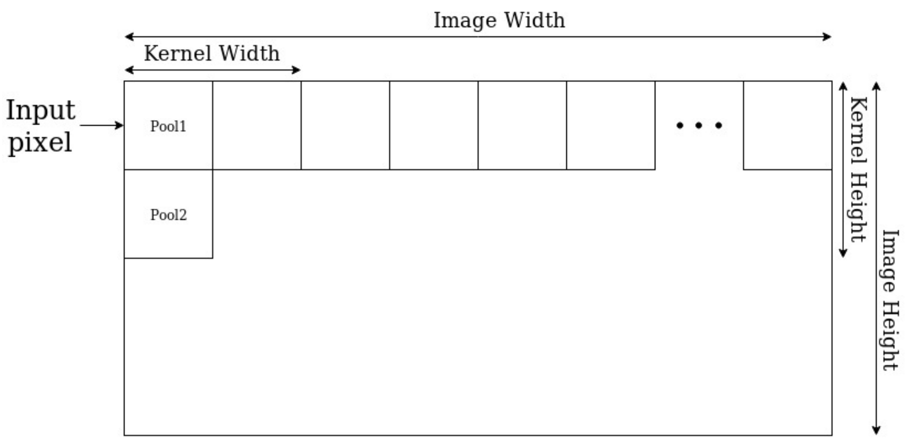


Figure 4.12: Data array poolV module.

values. Then it sends this values to the poolH module. Every 2 received values, it calculates the maximum of the values. In this way the data stream can be kept in the input while performing the necessary operations to continue the propagation in the network (figure 4.14).

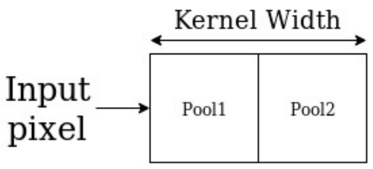


Figure 4.13: Data array poolH module.

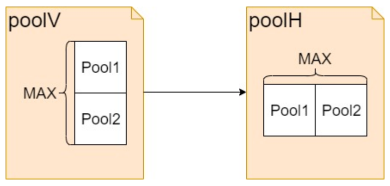


Figure 4.14: poolV and poolH modules.

4.6 Time Analysis

It was made the study of the time needed to travel the entire network in the FPGA. The image is streamed on CNN, so as long as there is enough data, processing is being done. When the last pixel in the image enters, the time required to finish the processing will be the time required to traverse the entire network. Network propagation time depends on the number of existing convolution and pooling layers plus the *InputLayer* (see figure 3.4). Each convolution layer has 3 main modules, *TensorExtractor*, *DotProduct* and *TanhLayer*. The latter is performed asynchronously, meaning no clock cycle is lost. Within *DotProduct*, 2 clock cycles are lost (1 cycle in each internal module). In Tensor Extractor 2 clock cycles are lost due to storage of data in an array. Thus, in each convolution layer, it occupies 4 clock cycles. The pooling layer contains 2 modules. Each one of these modules loses 1 clock cycle. In the end the pooling layer occupies 2 clock cycles. Thus, M being the number of convolution layers and N the number of pooling layers, the network propagation time is represented in clock cycles in equation 4.3.

$$Network\,propagation\,time = M * ConvLayer + N * PoolLayer + InputLayer \tag{4.3}$$

Chapter 5

Preliminary Results and Discussion

During this project several challenges were faced. Initially we had to draw the whole pipeline. Then we had to chose the architectures and protocols. When everything was ready to go, the problems encountered in HADDOC2 forced a restructuring of the original plan.

Finally, after all the study, development and patches, we were able to get a complete pipeline, from image acquisition, memory management, FPGA circuit control, CNN results and communication using the MQTT IoT protocol.

To test the entire pipeline, it was necessary to train a new version of the LeNet character detection network, using the MNIST dataset, because of the space limitation of the DE1-SoC board. Using only 2 convolution layers, followed by a tanh activation function and a pooling layer each (see figure 2.3) the network achieved a accuracy of 99% during training.

From table 5.1 we can see that even the reference implementation on CPU has issues with discerning between images 9 and 4, but correctly identifies the others. However, despite all the above mentioned corrections made to the HADDOC2 structure (convolution and pooling layers), the results on the FPGA are not all satisfactory. While the consistency and timing of the data flow pipeline was checked,

Image Number	Classification FPGA	Classification FPGA (%)	Classification CPU	Classification CPU (%)
1	2	50%	1	57%
3	2	88%	3	70.5%
4	3	92%	4	100%
5	3	99%	5	99.6%
7	2	61%	7	82.2%
9	3	92%	4	94.3%

Table 5.1: Classification results for the same input data using the FPGA and the CPU.

there remain errors that render the results useless. One of the possibilities is the weight conversion and also the normalisation along the network to avoid having degenerate computations. Having established an operational system end to end, from the sensor to the IoT, future work will have to double check the HADDOC2 revised modules to ensure numerical consistency of all computations.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work presents a heterogeneous system that combines HPS processing with the use of FPGA to deploy a CNN based system/device.The HADDOC2 framework which was intended to automatically generate the VHDL code from the CNN model, was restructured because it contained several flaws that prevented its use. A HADDOC2 support system was implemented on the FPGA together with the necessary modules needed to make it accessible to the management and controlling system running under linux. This system communicates to the internet by sending the processed data through the MQTT protocol. Besides the numerical aspects that steel need to be improved, a complete and functional pipeline is now available to generate an FPGA-based implementation from a Caffe CNN model.

One short paper and one demo have been accepted and presented at the REC'19 (XV Jornadas sobre Sistemas Reconfiguráveis) and Experiment@ International Conference 2019 (exp.at'19) respectively. Author versions of these articles are included as attachments to this document.

6.2 Future Work

The HADDOC2 framework has many features to improve. It has demonstrated that it is possible to have a fully CNN implemented in the FPGA, but the weight conversion has to be redesigned. Also this FPGA has approximately 500 kbytes of OCM. This is enough space to store all weights of a CNN, and be able to have larger networks running in the FPGA, or simply by using high-end larger FPGAs. Another aspect that needs attention is the pooling layer only accepts 2x2 filter size, and the convolution layer does not accept any stride. Weight conversion needs to be improved. As HADDOC2 creates code for a NN, regardless of target device this creates problems if FPGA space is small. Thus, multiplexing the convolution layers could be implemented to reduce the circuit design, keeping only one convolution layer in the FPGA, and loading the weights when necessary, so HADDOC2 could run on smaller platforms, using large CNNs. With the addition of these features and new activation functions, the capabilities of HADDOC2 will be increased by making a software capable of running CNN on the FPGA efficiently.

Another necessary improvement is to make HADDOC2 compatible with the new ONNX format, supporting NN models produced from different frameworks.

Bibliography

Abdelouahab, K., Pelcat, M., & Berry, F. (2018). The challenge of multi-operand adders in cnns on fpgas: How not to solve it! In *Proceedings of the 18th international conference on embedded computer systems: Architectures, modeling, and simulation* (pp. 157–160). ACM.

Abdelouahab, K., Pelcat, M., Sérot, J., Bourrasset, C., & Berry, F. (2017). Tactics to directly map cnn graphs on embedded fpgas. *IEEE Embedded Systems Letters*, *9*(4), 113–116.

Altera. (2018). Cyclone v hard processor system technical reference manual. Retrieved April 22, 2019, from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_54001.pdf

Bettoni, M., Urgese, G., Kobayashi, Y., Macii, E., & Acquaviva, A. (2017). A convolutional neural network fully implemented on fpga for embedded platforms. In *Cas (ngcas), 2017 new generation of* (pp. 49–52). IEEE.

Chen, Y. & Kunz, T. (2016). Performance evaluation of iot protocols under a constrained wireless access network. In *2016 international conference on selected topics in mobile & wireless networking (mownet)* (pp. 1–7). IEEE.

Du, L., Du, Y., Li, Y., Su, J., Kuan, Y.-C., Liu, C.-C., & Chang, M.-C. F. (2017). A reconfigurable streaming deep convolutional neural network accelerator for internet of things. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *65*(1), 198–208.

Intel. (2019). Embedded peripherals ip user guide. Retrieved April 30, 2019, from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., … Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (2006). A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, *27*(4), 12–12.

Molanes, R. F., Rodriguez-Andina, J. J., & Farina, J. (2017). Performance characterization and design guidelines for efficient processor - fpga communication in cyclone v fpsocs. *IEEE Transactions on Industrial Electronics*, *65*(5), 4368–4377.

Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., … Subhaschandra, S., et al. (2017). Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 acm/sigda international symposium on field-programmable gate arrays* (pp. 5–14). ACM.

Park, J. & Sung, W. (2016). Fpga based implementation of deep neural networks using on-chip memory only. In *2016 ieee international conference on acoustics, speech and signal processing (icassp)* (pp. 1011–1015). IEEE.

Patel, K. K. & Patel, S. M. (2016). Internet of things-iot: Definition, characteristics, architecture, enabling technologies, application & future challenges. *International Journal of Engineering Science and Computing*, *6*(5).

Pena, M. D. V., Rodriguez-Andina, J. J., & Manic, M. (2017). The internet of things: The role of reconfigurable platforms. *IEEE Industrial Electronics Magazine*, *11*(3), 6–19.

Raspa, N., Bacis, M., Natale, G., & D. Santambrogio, M. (2018). Condor, an automated framework to accelerate convolutional neural networks on fpga. Retrieved December 28, 2018, from https://necst.it/condor-convolutional-neural-networks-dataflow-optimization-using-reconfigurable-hardware/

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, *3*(5), 637–646.

Terasic. (2015). De1-soc user manual. Retrieved February 27, 2019, from https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=ae336c1d5103cac046279ed1568a8bc3

Terasic. (2019). De1-soc terasic web page. Retrieved August 27, 2019, from https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

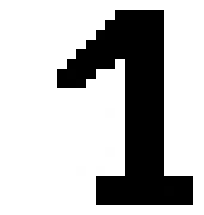
Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., & Vissers, K. (2017). Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 acm/sigda international symposium on field-programmable gate arrays* (pp. 65–74). ACM.

Venieris, S. I. & Bouganis, C.-S. (2017). Fpgaconvnet: A toolflow for mapping diverse convolutional neural networks on embedded fpgas. *arXiv preprint arXiv:1711.08740*.

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., ... Cong, J. (2017). Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th annual design automation conference 2017* (p. 29). ACM.

Appendix A

Images used in the CNN


(a)

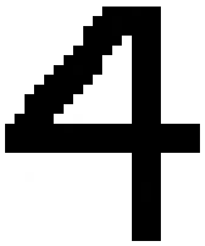
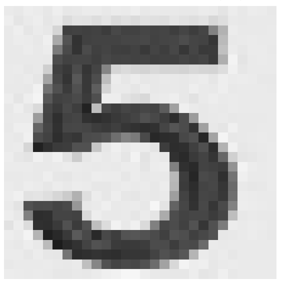
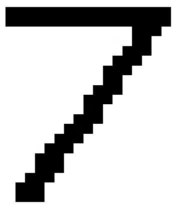
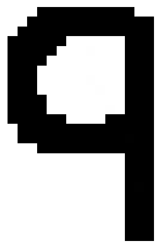
(b)

(c)

(d)

(e)

(f)

Figure A.1: Images used as CNN input.

Appendix B

Using the Communication Bridges in the DE1-SoC

This tutorial is intended for SoC devices (specific the DE1-SoC) and shows how to create a Quartus project to communicate between the HPS and FPGA using the HPS-to-FPGA bridges.

In this tutorial is used the *Quartus 14.1 Subscripted Edition*, but any other Quartus version that supports the board should work.

1. Install the *Quartus* software and the *Intel SoC EDS* that has specific libraries necessary to create the program.

2. Crete a blank project in Quartus.

3. Open Qsys with a open project, it will only have the clock. Save it with the name *FPGA_to_HPS_system*.

4. Add the *Arria V/Cyclone V Hard Processor System* from the *IP Catalog*, and keep the default settings.

5. Add 2 of *PIO (Parallel I/O)*, one with bitwidth = 10 and output with name *PIO_LEDR* and other with bitwidth = 10 and input with name *PIO_SWITCHES*.

6. Connect the clock to all components, and connect both *h2f_axi_master* and *h2f_lw_axi_master* to the PIO's.

7. In both PIO's go to external_connection in Export column and double-click to export. For the *PIO_LEDR* add the name *ledr* and for the *PIO_SWITCHES* add the name *sw*.

8. Go to *Address Map* tab and for the *PIO_LEDR* add the address 0x0000_0000 and for the *PIO_SWITCHES* add the address 0x0000_0020 in both bridges.

9. Now *Finish* and *Generate* the system.

10. Add the *.qip* file to the Quartus Project (go to *Project* → *Add/Remove Files in Project...* and add the file < *project_directory* > / < *name_of_qsys_project* > */synthesis/FPGA_to_HPS_system.qip*)

11. Set the *.qip* file as the *Top-Level Entity*

12. Add the pin assignments for the corresponding clock, reset, leds and switches (*Assignments* → *Pin Planner*)

13. In Quartus go to *Processing* → *Start* → *Start Analysis & Synthesis* and wait until complete

14. Now go to *Tools* → *Tcl Scripts...* and Run the script *hps_sdram_p0_pin_assignments.tcl* (its necessary because the *Arria V/Cyclone V Hard Processor System* block in Qsys has memory inputs and outputs, and this file generates the corresponding pin assignment)

15. Compile the Project

It is necessary to program the FPGA. This board when running with a linux OS, programs itself when turns on. So the programming file in the project folder is in < *project_directory* > / < *output_files* > / < *project_name* > *.sof*. This file has to be converted and added to the SD card. Following is a short tutorial:

1. In Quartus, go to *File* → *Convert Programminf Files...*

2. Add the .sof file in *Input files to convert* and in *Programming file type* select *Raw Binary File (.rbf)* and in *File name* change it to "soc_system.rbf". Click *Generate.*

3. Connect the SD card image to the computer. In the Shortest partition, should have a ".rbf" file. Substitute by the new file (if the file in the SD card has a different name then "soc_system.rbf", change the name of the new file to the name of the file in the SD card)

There is other simple way of do the conversion, and is shown bellow:

1. In the project folder go to $< project\_directory >$ / $< output\_files >$ and open a terminal here

2. Now execute the following: "quartus_cpf -c $< project\_name.sof >$ $soc\_system.rbf$"

3. Now in the same folder there is a new ".rbf" file, copy it to the SD card

Now its necessary a program to run on the HPS side to interact with the FPGA. The code presented bellow is written in C language. Starts by including the libraries, then define the "HW_REGS_BASE" that has control for a lot of peripherals, including the lightweigth bridge. Then define the normal bridge and the addresses for the leds and switches. After this the memory file is opened to map to the specific region of the lightweight bridge or the normal bridge (to change the bridges, check the commented lines in the code).

Save the program as "main.c". To compile is needed a cross-compile. Use the following code in terminal:
"arm-linux-gnueabihf-gcc -Wall -I /opt/altera/14.1/embedded/ip/altera/hps/
altera_hps/hwlib/include main.c -o de1_soc_hps".
Copy the generated file "de1_soc_hps" to the linux in the DE1-SoC board. Run it and use the switches in the board. It should change the corresponding led.

```
//=====================================================================
// This program uses the switches to control the leds in the FPGA
// To control the leds in the fpga, it makes use of the lightwight
// bridge, or the normal bridge
//
// Made by: Ricardo Barreto                    Date: 14/05/2019
//=====================================================================

#define DEBUG 1 // defines the shown information in the console

#include <stdio.h>      // printf()
#include <unistd.h>     // close() and usleep()
#include <fcntl.h>      // open()
#include <sys/mman.h>    // MAP
#include <signal.h>
//#include "hwlib.h"  // not used in this project
#include "socal/socal.h"
#include "socal/hps.h" // this file has the defined variable ALT_STM_OFST
//#include "socal/alt_gpio.h"   // not used in this project

//=====================================================================
// settings for the lightweight HPS-to-FPGA bridge
// The ALT_STM_OFST starts at 0xfc000000 and the HW_REGS_SPAN of
0x04000000 occupies all
the physical space until the end
// lightweight HPS-to-FPGA bridge starts at 0xff200000->ALT_LWFPGASLVS_OFST
#define HW_REGS_BASE ( ALT_STM_OFST )
#define HW_REGS_SPAN ( 0x04000000 )
#define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
//=====================================================================
//setting for the HPS2FPGA AXI Bridge
#define ALT_AXI_FPGASLVS_OFST (0xC0000000) // axi_master
#define HW_FPGA_AXI_SPAN (0x40000000) // Bridge span 1GB
#define HW_FPGA_AXI_MASK ( HW_FPGA_AXI_SPAN - 1 )
//=====================================================================
// Define the addresses of the leds and switches
#define PIO_LED_BASE 0x00000000
#define PIO_SWITCHES_BASE 0x00000020
//=====================================================================
volatile sig_atomic_t stop;
void catchSIGINT(int signum){
  stop = 1;
}

int main(int argc, char *argv[]) {
  // catch SIGINT from ctrl+c, instead of having it abruptly close
  signal(SIGINT, catchSIGINT);
  printf("Running. To exit, press Ctrl+C.\n\n");

  int mem_fd = open("/dev/mem", ( O_RDWR | O_SYNC ));
  if(mem_fd == -1) {
    printf("Can't open /dev/mem\n");
    return(1);
  }

  void *map_base_lw_bridge;
  // lw bridge
  map_base_lw_bridge = mmap(NULL, HW_REGS_SPAN, PROT_READ|PROT_WRITE,
  MAP_SHARED, mem_fd, HW_REGS_BASE + ALT_LWFPGASLVS_OFST);
  // normal bridge
  //map_base_lw_bridge = mmap(NULL, HW_FPGA_AXI_SPAN,
  PROT_READ|PROT_WRITE, MAP_SHARED, mem_fd, ALT_AXI_FPGASLVS_OFST);
  if(map_base_lw_bridge == MAP_FAILED){
    printf("Can't mmap\n");
    close(mem_fd);
    return(1);
  }

  volatile unsigned int *h2p_lw_led_addr = NULL;
  volatile unsigned int *h2p_lw_sw_addr =  NULL;
  h2p_lw_led_addr = (unsigned int *)(map_base_lw_bridge + PIO_LED_BASE);
  h2p_lw_sw_addr  = (unsigned int *)(map_base_lw_bridge + PIO_SWITCHES_BASE);

  unsigned int sw_val;
  sw_val = alt_read_word(h2p_lw_sw_addr);
  alt_write_word(h2p_lw_led_addr, sw_val);
  while(!stop){
        if(sw_val != alt_read_word(h2p_lw_sw_addr)){
            sw_val = alt_read_word(h2p_lw_sw_addr);
            if(DEBUG) printf("Switches changed!\n");
        }
        alt_write_word(h2p_lw_led_addr, sw_val);

        usleep(0.5*1000000); // usleep is in microseconds = 1*10^-6 seconds
    }

    if( munmap( map_base_lw_bridge, HW_REGS_SPAN ) != 0 ) {
        printf( "ERROR: munmap() failed...\n" );
        close(mem_fd);
        return(1);
    }
    close(mem_fd);
    return 0;
}
```

Appendix C

REC'19 - Extended Abstract

IoT Edge Computing Neural Network on Reconfigurable Logic

Ricardo Barreto, Jorge Lobo and Paulo Menezes

Institute of Systems and Robotics
Dept. of Electrical & Computer Eng.
University of Coimbra

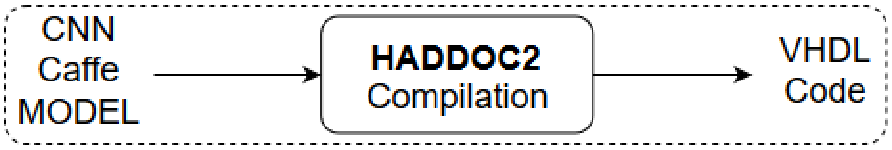
Abstract. This paper proposes the implementation of IoT devices that can communicate processed data obtained from local images. The image processing is done using convolutional neuronal networks (CNN). IoT devices follow the trend of using cloud computing to process the collected data. We intend a twist on the established paradigm and pursue an edge computing approach. Since we are targeting small and simple devices, we need some low power solution for the CNN computation. We also need to broke the IoT data. We will use a Terasic DE1 (SoC) reconfigurable system, with a field programmable gate array (FPGA), and hardwired ARM processor to build the IoT device. We will initially use the HADDOC2 framework that converts CNN models specified in Caffe to VHDL code. This will be integrated with the local IoT system on the SoC board. Building upon this initial solution, we will tackle how to address dimensionality issues and implement partitioning, pipelining, and multiplexing in time of the NN computations. In the end we expect to have a full toolchain to enable powerful IoT applications that rely on edge computing to only propagate to the network curated and useful data.

1 Introduction

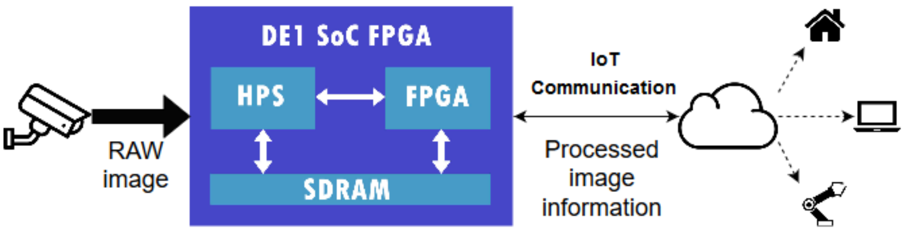
The search to make physical devices intelligent has always existed. Even when the first computer was developed, it was intended to draw conclusions and get results faster than the human could. Artificial intelligence presents itself as a tool capable of increasing the intelligence of current devices. CNNs appear as part of artificial intelligence aimed at recognising patterns in images, or videos. These networks are currently running on computers with large processing capacity, which are bulky and consume a lot of energy, which is a problem for remote and resource-constrained solutions. When creating an IoT device it needs to fulfil some requirements such as low energy, connectivity, reduced size [5], and developing custom circuits is often required. ASICs are custom-made chips for specific applications, having the advantage of being faster than FPGAs and are highly optimised. However they have great design and production costs. FPGAs are reconfigurable devices with the possibility of rapid prototyping, but also for end products. In this project we will use the FPGA due to the greater ease of programming and prototyping The design in FPGA can be a time consuming task. Thus, if we could have a program with the ability to produce VHDL code for the FPGA, the programming would be much simpler.

The HADDOC2 framework [1] has a tool chain that performs direct hardware mapping (DHM) CNN to VHDL. As shown in fig. 1a, the framework uses Caffe CNN models as input and generates the corresponding VHDL code to be synthetized and then deployed in the FPGA.

In fig. 1b we have a schematic of our high level architecture. A Terasic DE1 SoC development board device containing the FPGA and HPS will be used. The HPS running linux will be used to implement the IoT component, receive local external data, namely image or video, perform the communication with the outside and maintain the synchronisation of the data between the HPS and the FPGA. The FPGA will be used to implement previously synthesized CNN.



(a) HADDOC2 operation.



(b) High level architecture.

Fig. 1: General Architecture

CNNs tend to have multiple layers to obtain a precise result, and the more complex the problem to solve, the more layers the network will have. This increase in size leads to more operations and is a problem when implementing on FPGAs.

1.1 Related Work

Some frameworks to convert CNN models to hardware design language have been and continue to be developed. CONDOR [6] is an automated framework to accelerate CNN on FPGA. This framework is under development and the objective is to map CNN computation in hardware and take advantage of the parallelism of the CNN. Another framework is fpgaConvNet [7] that even automates the gathering of the target platform specifications, perform a custom mapping. However neither of the above are open-source, and since we want a base upon which to improve and customise, they were not used in this work. Also this program is not available to be changed. HADDOC2 is a framework that has the ability to convert CNN caffe models to VHDL code. Using DHM, HADDOC2 is taking maximum advantage of network parallelism by implementing it without any pipeline. So it takes full advantage of the FPGA. It is also not limited by the platform and the source code is open-source. This creates an advantage to developers to modify and improve the current HADDOC2 work.

Several works have explored FPGA implementations of neural networks (NN). CNN implementations have distinct aspects, but also a lot in common with NN. Simple NNs can be fully implemented on FPGAs with minor changes. As reported in [4] a NN is implemented and for the activation function 2 methods were tested. The first is a full implementation of the sigmoid function which uses multipliers and divisions, taking up a lot of resources. So in a second attempt a look up table (LUT) is used to replace the sigmoid function. With this technique some precision is lost, the memory access is slower, but a much more compact design is achieved.

Fully implementing a large NN on an FPGA can occupy all the reconfigurable logic. To overcome this situation a multiplexing approach is studied in [3]. This technique uses the large layer to create a control block that imports, at each layer, the respective neurons and weights. Using this method the FPGA is unimpeded to receive large networks, but the whole network will be slower because of the time multiplexing.

To explore in depth this technique, CNN are the main candidate as a consequence of its large size. As presented in [2] a CNN is implemented using a pipeline technique to implement the layers. The goal of the project was testing against common platforms used to execute neural networks, and the results are impressive. The FPGA got the same performance per watt as a 16 thread CPU Intel Xeon E5-2430. Comparing with the hardware used in embedded systems, a mobile GPU got substantial less performance then the FPGA.

2 Proposed Work

This work makes the use of the HADDOC2 framework to map to hardware a complete CNN on the FPGA. However this process is limited by the number of logic elements in the FPGA, so we will target smaller CNNs. The HADDOC2 has implemented some methods to optimise the final code. The number of multiplications is reduced by detecting redundant operations, multiplications by 0 removed, by 1 replaced by a single connection and by a power of 2 are transformed into shift registers. Although code created by HADDOC2 is DHM, this means that all neurons in a layer are mapped on the device to take advantage of inter-neuron parallelism, and each convolution is mapped separately and finally, each multiplier is instantiated separately. This method occupy a lot of hardware design. The CNN specification is given to HADDOC2 to obtain the VHDL code, that is synthetized for the specific hardware device. This is done offline, and only the final circuit is sent to the SoC hardware, along with the software for the IoT running on the HPS.

To feed the CNN an external camera connected to the DE1 board is used. This connection will be processed on the HPS side, running a program that will send the data from the camera to the FPGA. The communication between the HPS and the FPGA is an complex task. One way is the use a HPS to FPGA bridge, a high performance bus with 32, 64 or 128 bit. The other way is to make use of direct memory access (DMA). The later does not have a limit of bus width, and could easily transfer information between the HPS and FPGA system. The program running on HPS will also serve to maintain data consistency between the HPS and the FPGA.

The HPS program also includes the IoT communication to the web. The protocol used is the MQTT that relies on the TCP protocol for data transmission and includes an SSL certificate for security. This IoT protocol works in a publish-subscribe pattern, where publishers send a message to a topic, and external devices must subscribe to this topic if they want to receive the message. This enables a lightweight implementation, but secure and reliable. A broker is required on which to publish the messages. The HPS program is able to receive information too, by subscribing topics in the broker.

The HADDOC2 framework is limited by the strict DHM approach, but we want to go beyond and explore optimisations using the open-source framework. CNN processing goes through several layers. Image processing layers such as convolution, activation, and pooling are the most frequent in CNN processing. These require a large number of multiplications, which may limit implementations on FPGAs. These multiplications can be done using specific blocks or by creating the block with logic elements. Thus we could implement only one layer, and this be repeated in a cycle, multiplexing in time and maximise the space available on the FPGA. The network is no longer fully implemented, and may slow down processing, but it enables implementation of larger and more complex networks.

3 Expected Results and Conclusion

We intend to obtain an IoT device that combines high processing characteristics, keeping within a small size, and is energy efficient. For this we make use of a SoC that contains an FPGA and an ARM processor. The HADDOC2 framework converts the CNN model to VHDL and will be explored and modified to achieve the implementation of larger and more complex networks. IoT will be implemented on the ARM side of the processor using linux as the operating system. Thus a program will manage the communication with the outside, the connection to peripherals and the communication with CNN in FPGA.

References

1. K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry. Tactics to directly map cnn graphs on embedded fpgas. *IEEE Embedded Systems Letters*, 9(4):113–116, 2017.
2. M. Bettoni, G. Urgese, Y. Kobayashi, et al. A convolutional neural network fully implemented on fpga for embedded platforms. In *CAS (NGCAS), New Generation of*, pages 49–52. IEEE, 2017.
3. S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.
4. A. Muthuramalingam, S. Himavathi, and E. Srinivasan. Neural network implementation using fpga: issues and application. *International journal of information technology*, 4(2):86–92, 2008.
5. K. K. Patel, S. M. Patel, et al. Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges. *International Journal of Engineering Science and Computing*, 6(5), 2016.
6. N. Raspa, M. Bacis, G. Natale, and M. D. Santambrogio. Condor, an automated framework to accelerate convolutional neural networks on fpga, 2018.
7. S. I. Venieris and C.-S. Bouganis. fpgaconvnet: A toolflow for mapping diverse convolutional neural networks on embedded fpgas. *arXiv preprint arXiv:1711.08740*, 2017.

Appendix D

exp.at'19 - Demo

Edge Computing: A Neural Network Implementation on an IoT Device

Ricardo Barreto, Jorge Lobo and Paulo Menezes

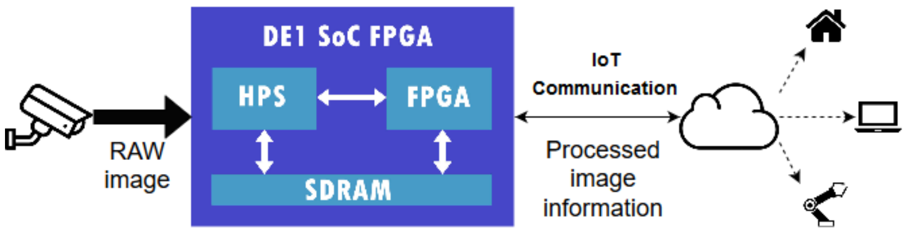
Institute of Systems and Robotics
Dept. of Electrical & Computer Eng.
University of Coimbra

Abstract. This demonstration showcases the use of reconfigurable logic to implement edge computing for IoT devices able to provide specific information from raw data produced from some sensor, e.g. a camera or microphone, instead of the raw data itself. In what concerns the embedded processing capabilities, the focus is image processing using convolutional neuronal networks (CNN). This approach is clearly distinct from the current trends in IoT devices of using cloud computing to process the collected data. We intend a twist on the established paradigm and pursue an edge computing approach. Since we are targeting small and simple devices, we need some low power solution for the CNN computation. The demonstration will be made on a Terasic DE1 (SoC) reconfigurable system, with a field programmable gate array (FPGA), and hardwired ARM processor to build the IoT device. The collected data from the CNN computation, is transmitted using an IoT protocol to a broker.

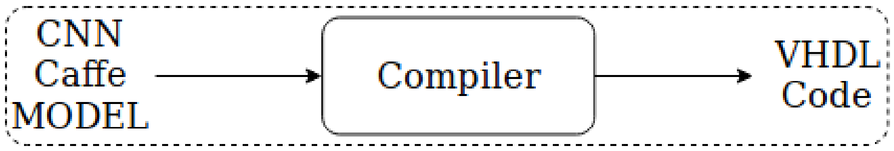
Keywords: Edge Computing, FPGA, IoT, CNN, ARM, HPS, Intelligent Devices

1 Introduction

The world is becoming increasingly intelligent as a result of technological developments and the expansion of the internet. In this way local devices as well as IoT devices are increasingly used to gather information but end up not performing any kind of data processing, leaving that task to cloud computing. Embedding AI capabilities on devices can be an interesting solution for improving their sensing capabilities (human-like) or performing in situation pattern recognition for producing e.g. semantic interpretations of the raw data, instead of the raw data itself. This leads to devices with a wide range of applications including monitoring, event detection, security systems, with the ability to interpret data locally, facilitating installation and communication. Among the AI techniques available neural networks and in particular convolutional neural networks have demonstrated very interesting capabilities in performing pattern recognition in images. These networks are currently running on computers with large processing capacity, which are bulky and consume a lot of energy, which is a problem for remote and resource-constrained solutions. When creating an IoT device it needs to fulfil some requirements such as low energy, connectivity, reduced size [6]. Therefore, an FPGA will be used because it is reconfigurable with the possibility of rapid prototyping, but also adequate for final products.



(a) High level architecture.



(b) Toolchain operation.

Fig. 1: General Architecture

2 Overview

2.1 Current Technologies

A conventional image processing system requires powerful processors, and typically data is obtained from one place, but processing is done elsewhere. Thus, by installing an intelligent system of video surveillance, monitoring, detection of events, etc. which makes use of a set of cameras, is quickly limited by the large bandwidth of the acquired data. In addition, the processing unit must respond to the amount of data to be processed, having high energy costs and

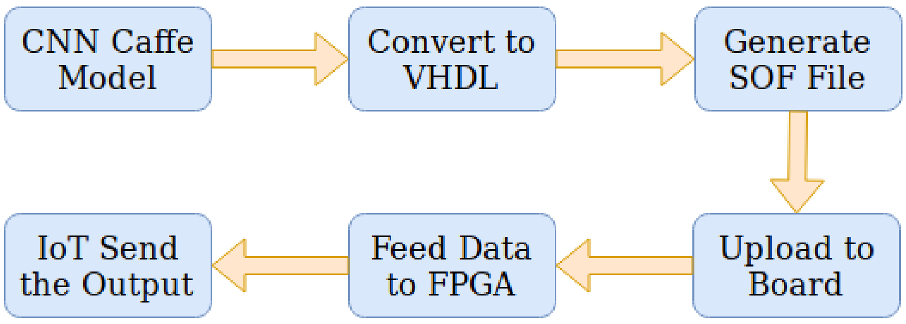


Fig. 2: Diagram of the steps to get the system working.

consumption. This system is also not flexible, requires a custom installation and configuration, and if cloud computing is used we may have problems of limited control, security and data privacy.

2.2 Advantages over Existing Technologies

Our system shows an IoT edge device where the data collected by the camera is processed locally using the hard processor system (HPS) and the FPGA. The intended CNN is converted using our toolchain and synthesized to the FPGA. The HPS runs a linux operating system with a dedicated program to collect camera data and maintains good communication between the HPS and the FPGA and the outside world using the MQTT protocol for IoT devices. Only the final result of CNN's image processing is sent, occupying a minimum bandwidth compared to the image or video transfer. In this way we have a system with local processing, safe, modular and low power.

3 System Operation

As seen in figure 1a this system collects images, processes them and sends the final result. Both HPS and FPGA can be used to process CNN. To perform the processing on the FPGA the chosen CNN needs to be converted to VHDL code. The HPS being a low-power ARM processor, it can only run simple CNNs, yet getting results locally. But for better efficiency and performance, a custom circuit is used on the FPGA to accelerate the CNN, achieving faster processing.

This process is simplified using an of the shelf toolchain represented in 1b where a CNN Caffe model provides the input and outputs the corresponding VHDL code. After this process the CNN code is integrated in our project, that wraps up the communication modules in linux on the HPS and implements the interface on the synthesised circuit placed on the FPGA. On the HPS side there is a program that manages the system, sending the image to the FPGA using the memory of the board and communicating with the FPGA using direct acess memory and a small bridge of direct communication (figure 3). Finally the HPS receives the data processed in the FPGA and finalises the calculus of the CNN.

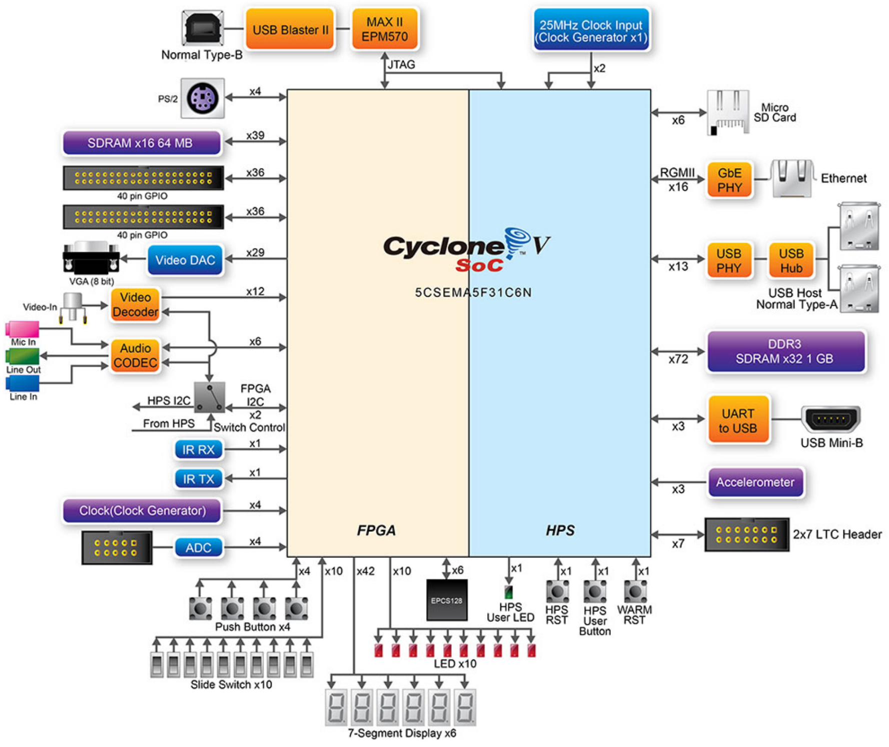


Fig. 3: Terasic DE1-SoC diagram.

Since the board has internet communication, the MQTT protocol is used. This protocol works in a publish-subscribe pattern, where publishers send a message to a topic, and external devices must subscribe to this topic if they want to receive the message. Thus only

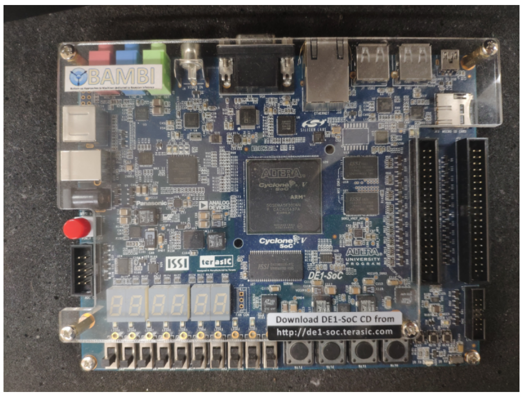


Fig. 4: Terasic DE1-SoC development board.

a message is sent using a very low bandwidth. This data process is shown in greater detail on the diagram in figure 2.

3.1 Demonstration

A Terasic DE1-SoC development board (figure 4) containing both the HPS and the FPGA is used in the demonstration of this project. A CNN trained on caffe is previously converted to VHDL code. This CNN is accelerated using the FPGA. Linux on the board supports the CNN running on the FPGA by sending and receiving the data.

The board behaves always as an IoT device, performing the local processing and sending the results to the internet. Converting the CNN Caffe model to the circuit requires a powerful computer system with the the toolchain installed, but this is done offline in the development stage. Synthesis of the VHDL code is performed using Quartus II. This is a program that needs some processing power, but since this process only happens once, it does not have a negative effect if it takes more time.

4 Conclusion

We intend to demonstrate an IoT device that combines high processing characteristics, keeping within a small size, and is energy efficient. For this we make use of a SoC that contains an FPGA to perform image processing using CNN and the ARM processor to run a software that will feed the FPGA with data and keep the synchronism of the tasks and communicate with the internet. With this approach we have a modular edge computing device with the ability to adapt to various uses, based on CNNs, with a performance equivalent to a computer, but with the characteristics and limitations of an embedded system IoT device.

References

1. K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, and F. Berry. Tactics to directly map cnn graphs on embedded fpgas. *IEEE Embedded Systems Letters*, 9(4):113–116, 2017.
2. M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva. A convolutional neural network fully implemented on fpga for embedded platforms. In *CAS (NGCAS), 2017 New Generation of*, pages 49–52. IEEE, 2017.
3. Barreto, Ricardo and Lobo, Jorge and Menezes, Paulo. IoT Edge Computing Neural Network on Reconfigurable Logic. In *Proceedings of the 15th Portuguese Meeting on Reconfigurable Systems*, 2019.
4. S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.
5. A. Muthuramalingam, S. Himavathi, and E. Srinivasan. Neural network implementation using fpga: issues and application. *International journal of information technology*, 4(2):86–92, 2008.
6. K. K. Patel, S. M. Patel, et al. Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges. *International Journal of Engineering Science and Computing*, 6(5), 2016.