POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Low power implementation of neural network extension for RISC-V CPU

Supervisors

Prof. Danilo DEMARCHI

Ronan BARZIC

Vemund BAKKEN

Candidate

Dario LO PRESTI COSTANTINO

April 2023

A mamma e papà
che mi hanno insegnato
cos'è l'umiltà,
la spontaneità,
l'amore.

Abstract

Deep Learning and Neural Networks have been studied and developed for many years as of today, but there is still a great need of research on this field, because the industry needs are rapidly changing. The new challenge in this field is called *edge inference* and it is the deployment of Deep Learning on small, simple and cheap devices, such as low-power microcontrollers. At the same time, also on the field of hardware design the industry is moving towards the RISC-V micro-architecture, which is open-source and is developing at such a fast rate that it will soon become the standard. A batteryless ultra low power microcontroller based on energy harvesting and RISC-V microarchitecture has been the final target device of this thesis. The challenge on which this project is based is to make a simple Neural Network work on this chip, finding out the capabilities and the limits of this chip for such an application and trying to optimize as much as possible the power and energy consumption.

To do that TensorFlow Lite Micro has been chosen as the Deep Learning framework of reference, and a simple existing application was studied and tested first on the SparkFun Edge board and then successfully ported to the RISC-V ONiO.zero core, with its restrictive features. The optimizations have been done only on the convolutional layer of the neural network, both by Software, implementing the Im2col algorithm, and by Hardware, designing and implementing a new RISC-V instruction and the corresponding Hardware unit that performs four 8-bit parallel multiply-and-accumulate operations. This new design drastically reduces both the inference time (3.7 times reduction) and the number of instructions executed (4.8 times reduction), meaning lower overall power consumption.

This kind of application on this type of chip can open the doors to a whole new market, giving the possibility to have thousands small, cheap and self-sufficient chips deploying Deep Learning applications to solve simple everyday life problems, even without network connection and without any privacy issue.

Keywords

Artificial intelligence, Deep learning, Neural networks, Edge computing, Convolutional neural networks, Low-power electronics, RISC-V, AI accelerators, Parallel processing

Acknowledgements

I would like to thank ONiO (in particular the CEO Kjetil Meisal and the CTO Vemund Bakken) for having given me the possibility to be part of this company and feel the great vibes and hard work that they put on their innovative design. A special thank you goes to my technical supervisor at ONiO, Ronan Barzic, who has been a real mentor to me and was always ready to help without hesitation.

I am also very grateful to the Professors that guided me as double degree student in KTH, Stockholm: Carl-Mikael Zetterling and Masoumeh Ebrahimi. They followed my work since the beginning and gave me great tips along the way. I would like to thank my thesis supervisor from Politecnico di Torino as well, Danilo Demarchi.

I would like to thank the IEEE, Elsevier and O'Reilly Media, Inc. for having given me the permission to reproduce the Figures present in their paper, articles and books.

This thesis represents the end of a 5 year journey that brought me from being a child in a small town in Sicily to being a young engineer who already entered the labour market and experienced different cultures and lifestyles around the world. If I think about it, it does not even feel real how much I have changed and learned from a human, social and of course technical level, and all that has only been possible thanks to the people that surrounded me during all these 24 years, who really made everything worth it.

First of all, I would not be here without the constant and immeasurable love, support and understanding of my family: my mom Laura, my dad Hunynush and my grandma Aba above all. My brother Paolino has been precious in these years as well, being my constant reference and example that taught me that everything is possible when you dare and put effort, but he also taught me to step out of my comfort zone and experience new realities to truly understand who I am and what I want, even though I still have to get to that point.

Another fundamental part of my travel is represented by the ones that were always there when I came back home, who became home themselves: the friends of a lifetime Piero, Nino and Alessandro, as well as my high school friends Paolo and

Stefano, together with Riccardo, one of the closest if not the closest person to me in these years, who taught me what friendship means in its authentic essence.

As this is the conclusion to my path, I would like to thank all the people that have been close to me and helped me in any way to feel home and feel good wherever I was along those years: Stefano and Daniele, with whom I started my “adventure" at PoliTO; the “Siciliani onesti" group (Luca, Andrea, Ettore, Giovanni and Antonio) that brought the best Sicilian vibes up to Turin; Simone, Filippo and Nikola, who shared the greatest American experiences with me in Arkansas and all the friends people from the second floor of Collegio Einaudi, sezione Corocetta, that made me truly enjoy the years spent in Turin, even during lockdown.

The final part of this journey has been in Scandinavia, between Sweden and Norway, and in particular in Stockholm I met some of the best people that made this experience just unforgettable and special, being close to me in the good times and extremely important in the bad ones: first of all the “Tazzoni + 1" group, but also Vittorio, Gianluca, Simone, Edoardo, Luca, Giacomo, Ludovica, Ilaria, Lucia, Joe, Javi and many others that I do not mention here but are vividly present in my memories. Here I want to thank especially one person that quickly became and still is one of the closest and most authentic friends I have: Federica, with whom I shared almost all the best memories in Stockholm and who is always ready to listen and help and really managed to put a smile on my face even in the worst moments, teaching me to always put the others first and that being sweet is one of the most important qualities a person can have.

At last, I would like to thank the person who, despite everything, represented for all those 5 years my main harbor, anchor and shelter, who gifted me with the constant peace of mind and stability that has been the secret of my success, who taught me what it means to truly and sincerely love without conditions and who made me reach, as much as possible, that rare and fleeting glimpse that someone calls happiness: to Chiara.

Table of Contents

List of Tables

List of Figures

Acronyms

ADC	Analog to Digital Converter
AFU	AI Functional Unit
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
API	Application Programming Interface
APU	AI Processing Unit
AWS	Amazon Web Services
BSP	Board Support Package
BYOC	Bring Your Own Codegen
CISC	Complex Instruction Set Computer
CNN	Convolutional Neural Network
CNTK	CogNitive ToolKit
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CSR	Control and Status Register
DLA	Deep Learning Accelerator
DLR	Deep Learning Runtime
DL	Deep Learning
DSP	Digital Signal Processing
DWT	Data Watchpoint and Trace
ELL	Embedded Learning Library
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit

GAS	GNU ASsembler
GCC	GNU Compiler Collection
GEMM	GEneral Matrix Multiply
GPIO	General Purpose I/O
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HAL	Hardware Abstraction Layer
HW	HardWare
I2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction Set Architecture
ITM	Instrumented Trace Macrocell
JTAG	Joint Test Action Group
MAC	Multiply and Accumulate
MAE	Mean Absolute Error
MCU	Micro Controller Unit
MEMS	Micro-Electro-Mechanical Systems
MISO	Master In Slave Out
MLF	Model Library Format
ML	Machine Learning
MMU	Memory Management Unit
MOSI	Master Out Slave In
MSB	Most Significant Bit
NN	Neural Network
NPU	Neural Processing Unit
ONNC	Open Neural Network Compiler
ONNX	Open Neural Network eXchange
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMSE	Root Mean Squared Error
RTL	Register Transfer Level

RTOS	Real Time Opertaing System
ReLu	Rectified Linear unit
SDG	Sustainable Development Goal
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SW	SoftWare
TE	Tensor Expression
TF	TensorFlow
TVM	Tensor Virtual Machine
UART	Universal Asynchronous Receiver-Transmitter
UN	United Nations

Chapter 1

Introduction

This chapter describes the specific problem that this thesis addresses, the context of the problem, the goals of this thesis project, and outlines the structure of the thesis.

Nowadays the applications of AI and specifically of ML are growing in number at a very fast rate and touch almost all the fields of technological progress. The focus of the research on ML is at its peak and the impact of the new results is already visible in several use-cases, from new medical diagnosis instruments to common smart devices for the modern domotics. While the large scale applications based on cloud computing are already quite known and have been heavily developed in the recent years, on the other hand the trend now is to move towards the small scale, trying to implement similar simpler solutions in the millions of embedded devices that truly are the protagonists of the modern technology.

At the same time, in the field of embedded devices and microcontrollers, in the recent years there has been the tendency of moving everything to the so called *open source* development, meaning that it is the community that creates and develops the product, granting free access to all the details and design ideas to everyone. In particular, the design of the hardware itself can now be open-source thanks to RISC-V, which is an open-standard ISA that can replace the expensive ARM designs for microcontroller cores.

This thesis project originates from this two research trends and tries to unify them by taking a typical ML embedded application and implementing it on a custom low power batteryless RISC-V core designed by ONiO, the company where the project has been carried out, combining the challenges outlined above with the restrictions coming from such an extreme low power design.

1.1 Background

The background for this project is quite broad and goes from the AI field in general to the microarchitecture design and RISC-V ISA, considering all the related areas that allow for better efficiency and lower power consumption.

On one hand, starting from the AI it is possible to narrow down the focus to the so called **edge inference** and to TF Lite Micro, the ML framework adopted in this project for the development and deployment of the application.

On the other hand, again starting from a general overview of the RISC-V ISA, it is possible to focus only on the instruction types and extensions needed for the specific case of this project, as well as the HW design methods that allow an efficient and low power implementation of the operations.

1.2 Problem

Several examples and applications of ML edge inference can be already found in literature, as well as thousands of RISC-V projects and low power designs. Moreover, some studies and applications have been also carried out about edge inference on RISC-V custom cores or accelerators, therefore the challenge that makes this thesis project different and in some ways original is related to the extreme restrictions that a microcontroller powered only by energy harvesting imposes, and how to match these constraints with the requirements coming from edge inference of a NN, however simple it is.

ONiO.zero is the custom batteryless microcontroller produced by ONiO and it will be the target device for this thesis project. The final research question is therefore: is it possible to perform edge inference on the ONiO.zero? Which HW and SW approaches can be used to make it possible and efficient enough? What are the limits of ONiO.zero in this specific application and how can they be overcome though HW/SW optimizations?

1.3 Purpose

The higher level purpose of this thesis project is what truly makes it exciting and fascinating, because it is profoundly innovative from all points of view: cheap and small devices that need almost no maintenance being batteryless and therefore self-sufficient are already a cutting-edge technology that could become the standard in the next years, but being able to exploit ML algorithms on those chips would open the doors to new possibilities and applications in any field that could change how people do things.

Moreover, the real application of such a project would also guarantee several other benefits from a sustainable and ethical point of view. As a matter of fact, having a self-sufficient chip means getting rid of batteries, which are a great cause of pollution both for their production and for their disposal, while at the same time being able to perform ML edge inference means to avoid the need of cloud computing, therefore all the collected data will be stored and processed only on the device itself, which solves all the privacy and social issues.

1.4 Goals

The goals of the project can be directly derived from the problem statement, even if the project has been carried out having sequential objectives depending on each other. The final objectives from a general point of view can be summarized as follows:

- Choose an existing ML simple application and port it to the constrained RISC-V core. Here some subgoals can be defined.
  - Test the NN as it is on a general purpose board equipped with an ARM core (SparkFun Edge board).
  - Quantize the model and change the application accordingly to meet the restrictions of the RISC-V core and test it on the SparkFun Edge board.
  - Port the application to a constrained RISC-V core by rewriting the operators and test it on the FPGA model of the core.
- Find out where and if the process can be improved towards a lower power consumption and a better efficiency. This can be done both by HW and by SW, by adding some custom instructions, designing a specialized HW accelerator or adding a AFU in the core pipeline.
- Take measurements to find out the limits and the capabilities of the constrained RISC-V core when implementing such an application.

1.5 Research Methodology

The methodology adopted in this project tries to move step by step from a very simple application to the final optimized HW design. Given the initial limited familiarity with the AI topic, this kind of method is essential in the present case, because it allows to fully understand the basics first, and then try to apply them in practice little by little.

As a consequence, the steps involved in this project are the following:

- Generic and detailed literature research to understand what is a NN, how to develop and deploy it and how nowadays it is implemented on microcontrollers.
- Generic and detailed literature research to see how a RISC-V core can be optimized for few specific operations.
- Choice of an AI framework and of a simple application that will constitute the SW core of the project.
- Develop and test the simple application on a common microcontroller equipped with an ARM core.
- Optimize the application in size, memory and inference time for the ARM core to see what is needed, how it works and be prepared for the RISC-V optimizations.
- Port and test the entire application to the RISC-V custom core.
- Optimize the HW and the SW to achieve an acceptable power consumption and find the capabilities of the custom RISC-V core in deploying a NN application.

As it can be seen, the actual work on the RISC-V HW represents only the final step, confirming once more the idea of incremental approach adopted in this project. Of course more time and focus is put on the two last steps, but it is important to understand that the previous ones are fundamental to have a concrete idea of how the whole system works and this cannot be achieved with a different methodology that, as an example, starts directly from the RISC-V implementation and tries to optimize it without knowing how different HW implementations work.

1.6 Delimitations

The work done in this thesis project is first of all focused on the low power HW perspective, even though at a first look it might seem that the central topic is the design and development of a specific NN model. It is therefore important to clearly state that the design, training and quantization of the NN model, as well as its inference on an ARM core are not part of the design goals of this project, but rather they are essential integration steps that lead to the RISC-V low power HW implementation, which represents the real purpose. This explains why a simple ready-made NN model has been chosen as reference starting point, without

spending too much time modifying it, as well as the integration on the ARM core has been done following a simple default approach.

One more thing to point out is that, as stated in the previous paragraphs, the final purpose of the project is to explore and find out about the capabilities and the limits of a highly constrained RISC-V core, therefore it is not needed to reach a certain performance level or to make the final application work effectively in a real case. This is why the results of this project should be seen from a research perspective, since here a new custom core with very restrictive characteristics is being used for the first time for such an application.

1.7 Structure of the thesis

Chapter 2 presents relevant background information about AI frameworks, convolutional NNs, RISC-V and the corresponding applications in literature. Chapter 3 presents the methodology and method used during the progress of the project, describing the main general choices made before starting with the implementation part, both from the SW and HW point of view. Chapter 4 details the technical work done during the project, meaning every step from the ready-made example on the SparkFun Edge board to the final RISC-V optimizations, with all the related tests and SW/HW changes. Chapters 5 and 6 contain the obtained results and the following considerations about the measurement outcomes. Finally, chapter 7 wraps everything up, making some conclusions on the work done and suggesting some possible paths for further development of the project in the future.

Chapter 2

Background and Related Work

This chapter tries to highlight the necessary background knowledge that allows to understands the method and the results of this project. As already stated, there are mainly two areas on which the concept of the thesis is based: RISC-V microarchitecture and ML, more specifically NN. As a matter of fact, the two Sections of this chapter focus on ML (Section 2.1) and RISC-V (Section 2.2), trying to give a brief general overview and go more and more in detail when it comes to the actual low level topics of the thesis. As a consequence, Section 2.1 focuses more on the CNN structure and on the specific framework chosen to develop the application (TF Lite Micro), while Section 2.2 analyzes the microarchitectural approaches to develop RISC-V core. Finally, Section 2.3 highlights the most relevant examples found in literature about projects related to these two fields.

2.1 Machine Learning and Neural Networks

Nowadays AI, ML and NN are used more and more frequently in the literature and in the research, but it is not that common to know what makes them different and what is common between them. The more general concept is surely related to the AI, which allows the computers to achieve some goals in the way humans do, and this AI field actually encompasses the other two.

ML is a field which allows computers to adapt to certain situations and react to changes. This is made possible by the *training*, during which the computer actually learns how to behave in a specific task with variable parameters and input data. As a consequence, training means feeding the machine with numerous known input data all different from each other, with the objective of making the machine's

output as close as possible to the ideal known output, so that in case of new, unknown input data, the output has a high probability to be very close to the ideal one. The known input data set is applied to the network many times and each time is called *epoch*.

Neural Networks can be seen as a sub-field of Machine learning and are sometimes referenced as DL. They try to mimic in a simple way how the human brain works, i.e. through neurons and synapses. This explains why it is "deep", because it is possible to use several layers of neurons to process the data before having an output. Each layer will have the goal to identify the presence of some features (or particular patterns) in the data given at its input, so the deeper layers will work on the features found by precedent ones, trying to find higher level features. The final goal will be to see if the highest level feature found corresponds to any of the object classes learned through training.

To understand how everything works, it is just needed to know that in a simplified way, the information in the human brain is processed by the neurons, connected through synapses, which are the link between the output of the previous neuron and the input of the following one. In neural networks field, the output of a neuron is called activation and the synapses are called weights. At each layer, every neuron will perform a weighted sum of all its inputs and then apply a non-linear function to the result (called activation function), with the purpose of finding some sort of feature in the data. At each layer of neurons higher level features can be identified and the final output will be a vector of scores that tells how high is the probability of having found a specific feature between the known ones.

As an example, at each layer of the network, the formula applied by a neuron is the following (eq. 2.1, from [1]):

$$y_i = f(\sum_{i=1}^{n} W_{ij} \times x_i + b) \tag{2.1}$$

with W_{ij} being the weights (synapses), x_i being the previous layer neurons' activations, f is the non linear function and b is a bias term. Figure 2.1 (retrieved from [1]) describes and explains the formula.

As a consequence, designing a neural network model means choosing how many layers of neuron to have, how many neurons to have in the internal (called "*hidden*") layers and what kind of activation function each layer will have. The number of neurons at the input layer is usually given by the number of inputs (measured values) and the number of neurons on the output layer are given by the wanted outputs (features to be found).

There are many different kinds of NNs that could be more or less convenient depending on the type of data to be processed at the input and depending on how

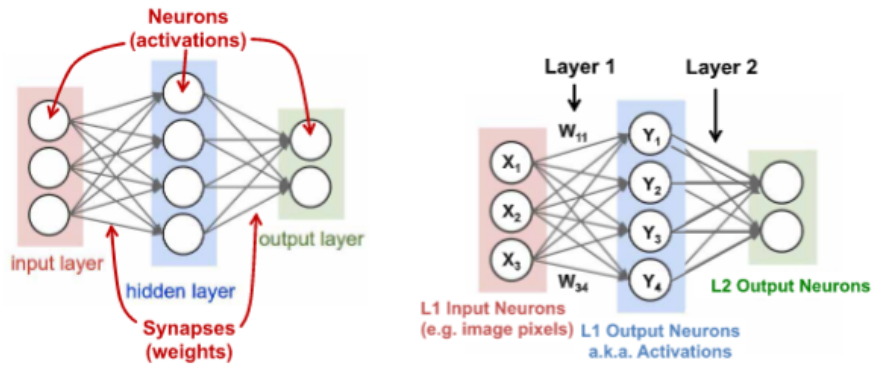


Figure 2.1: Neural network functioning example [1]. ©2017 IEEE

this data can be processed in the easiest and fastest way to recognize the wanted features.

It can be now easier to understand what training means in this context: changing the weights in order to minimize the loss, which is the difference between the current and the ideal scores. Again, when designing a neural network model, part of the design is also the choice of the training algorithm, meaning the function(s) that control how the weights are updated during the training process, in order to minimize the loss.

Another important term in this field is *inference*, which is simply the deployment of the neural network on a practical case, so implementing it on a machine, feeding it with an input and getting an output of scores.

2.1.1 Convolutional Neural Networks (CNNs)

One specific type of NN is the CNN, which is one of the most used types in many recent applications, for its great results and for the possibility to efficiently accelerate its operations. It fits really well when it is needed to find some sort of relationship between adjacent values, in order to extract particular features, and that is why most of its luck comes from the image recognition and classification, where it is used at pixel level, but it can give great results also in many other fields, such as speech or gesture recognition, if the input data is pre-processed in the proper way.

In order to recognize these patterns, this kind of network is organized in subsequent layers, and in each layer the chosen features are recognized by the use of filters that are applied to the input data through the convolution operation. In the first layer the filters will only recognize simple features between immediate neighbors, and pass the result to the successive layer, where other filters can see if

the features found in the first layer can be composed together to form some more complex features, and so on, until the final complex features can be recognized in the last layer.

Each layer has a specific number of filters, where each filter will be able to recognize a particular feature and is composed by the *weights* that are tuned during training. Each filter will be convoluted with the input and it will give as a result a *feature map*, which indicates where the feature has been found, so after the first layer there will be as many output feature maps as the layer's filter number. Each of these output feature maps could have the same dimension as the input data or not, depending on how the filter is applied to the input data, and this is decided by two convolution parameters: the *stride* and the *padding*. The stride value controls how the window is moved along the input matrix, so stride=1 means that the window is moved of just one element at a time both in the vertical and horizontal direction, while with stride=2 the window will move of two elements in both directions, so in the end less operations will be performed and the output feature map will have less elements. The padding controls if some 0-padding is applied to the input feature map so that the window can be always centered on the input matrix value. It is actually very common that the shape is the same, provided that stride is 1 and a padding is made in the corner cases for the filter values that are out of the input data. an example is given in Figure 2.2.

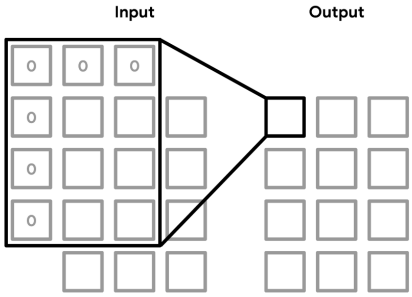


Figure 2.2: CNN filter multiplication: padding in the corners [2].

Another common layer that usually follows the convolutional layer is the Max-Pool, which has the goal to shrink the size of the output feature maps to make the computation easier and gradually go to high level features eliminating useless data. Its functioning is very simple, it splits the feature map into non overlapping windows and it passes to the new output only the maximum value of that window (see Figure 2.3).

The following step is the last one before a new convolutional layer is used: it is the *dropout*, which perform the so called *regularization*. It is a way to avoid overfitting, meaning that the model could fit too well the training data, but will not be able to withstand unexpected noise or strange behaviors of the input. During

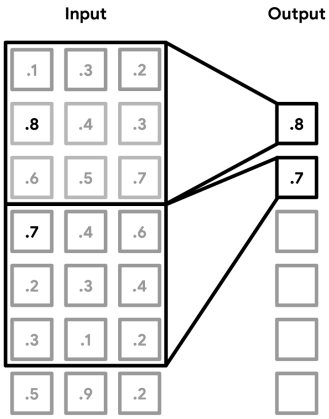


Figure 2.3: CNN MaxPool layer functioning [2].

dropout, some of the output values are randomly set to 0, losing in this way their information. In this way the model is also trained for noisy environments. Of course, this layer is only applied during training, since it would be counterproductive to set some output values to 0 during inference, losing in this way precious information about the unknown input.

After that, new layers formed by convolution, MaxPool and dropout can be attached based on the needs, designing the actual NN.

It is worth it to analyze in detail the convolutional layer to fully understand how it works. The input data to each conv layer are the input matrix (or matrices) and the filters. Each input matrix can have more than one so called *channel*, or *feature map*, ant it is important that every single filter has the same number of channels as the input matrix. An example of channels can be found in the image processing, where the same image typically has three channels (RGB). In that case every filter will have three channels as well.

For every single filter, all filter's channels are convoluted with the corresponding input channels (feature maps) and the results are summed together to have only one output feature map per filter. With the term *convoluted* it is meant that a window with the same same dimensions of the filter is applied to the input channel's matrix (with an eventual padding), and all the input values that are inside the window are multiplied pointwise with the correspondent value of the filter's channel and then summed together to generate a scalar. This is repeated for all the input channels (convoluted with the correspondent filter channels) and all the produced scalars related to the same filter (but to different channels of that filter) are finally summed together to generate the output value stored in the output channel correspondent to that filter. What desscribed above is related to just one element of the input matrix (with all its channels) and produces just one element of one channel of the

output matrix. The other channels of the output are generated with the same procedure, but with a different filter (with all its channels).

Figure 2.4 shows how the convolution is typically performed, with 2 input matrices each with 3 channels (or Input Feature maps) and 2 filters (each with 3 channels). The number of outputs is the same as the number of the input, but each output will have as many channels (or Output Feature maps) as the number of filters.

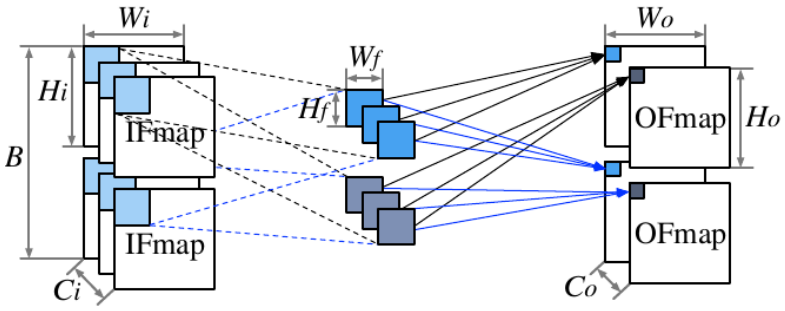


Figure 2.4: Schematic view of CNN layer functioning [3]. ©2019 IEEE

The corresponding mathematics is summarized in the following Equation 2.2 (from [3]), where C is the number of input channels, H and W are the height and width of the channels or of the filters (with _i being input and _f being filter). Note that this equation does not consider the eventual padding made to have input and output feature maps of the same dimension.

$$OFmap\,[k]\,[x]\,[y] = \sum_{c=0}^{C_i-1} \sum_{r=0}^{H_f-1} \sum_{s=0}^{W_f-1} Fil\,[k]\,[c]\,[r]\,[s] * IFmap\,[c]\,[x+r]\,[y+s] \quad (2.2)$$

A simple example can be seen in Figure 2.13. Looking at the second Conv2D layer, it has one input with 8 channels and it uses 16 filters. Each filter needs to have the same number of channels as the input, therefore 8 channels in this case, while the output will have as many channels as the number of filters, so 16 here. The operations done in this case can be summarized as follows: for each filter every channel is separately applied to the correspondent input channel, and each result is then summed pointwise across all the channels, so that one filter will generate one output feature map. The same procedure is repeated for every filter with its channels.

In most cases the NN needs to end with one or more fully connected layers, to go from feature maps to being able to see if the sequence of recognized features forms the wanted object, and give the related *scores*, or class probabilities. To do so, all the output feature maps of the last convolutional layers are flattened

together in just one long 1D vector of data, which gradually shrinks to only N output values thanks to a series of fully connected (or dense) NN layers, where N is the number of object classes to be recognized. It is important to highlight that each convolutional and fully connected layer has its own activation function. The last fully connected layer should have as outputs some actual probability values that can be understood by humans, and this is why it has a softmax activation function.

CNN popular optimization: Im2col

One of the several ways to optimize the computation of a convolutional layer is called *Image to column*, or *Im2col*, which basically consists in transforming the convolution operation into a common matrix multiplication. This is very useful in case the hardware is highly optimized for matrix multiplication, such as on GPUs. In many other cases, the common optimized matrix multiplication is done through the so called GEMM algorithm. The Im2col algorithm first takes the filter elements and puts them in a column, then it does the same with all the channels of that filter stacking all these values in just one long column. This is repeated for every filter, having one column for each filter (with all its channels). Then the elements of the input matrix contained in the convolution window are placed in a row and the same is done for every channel of that input, creating one long row related to that specific window. Similarly, considering how the window moves (stride and padding), all the rows related to every possible window are stacked one on top of each other. In this way a new input matrix is created as it can be seen in Figure 2.5, where looking at the colors is very clear how the elements are placed and how is it possible that a simple matrix multiplication will give the complete output.

All the symbols in the Figure have the same meaning as in Equation 2.2. It can be seen how some elements will be duplicated in this new matrix (elements in a red-dotted box), but this is needed to make the computation easier.

The same procedure can be done by transforming the filters into rows and the image into columns (that is why the name of the procedure is *Image 2 Columns*).

2.1.2 Neural Networks in Low Power applications

As it was described in the background Section, having a set of neurons means performing many weighted sums and applying a non-linear function to them several times, therefore neural networks require high power consumption. This leads to the fact that most of the computation can be reduced to matrix multiplications, and this explains why GPUs are widely used for these kinds of applications, together with several MAC units in CPUs. In many cases this is not a problem, since it is possible to perform inference in powerful, well equipped devices, or in devices that

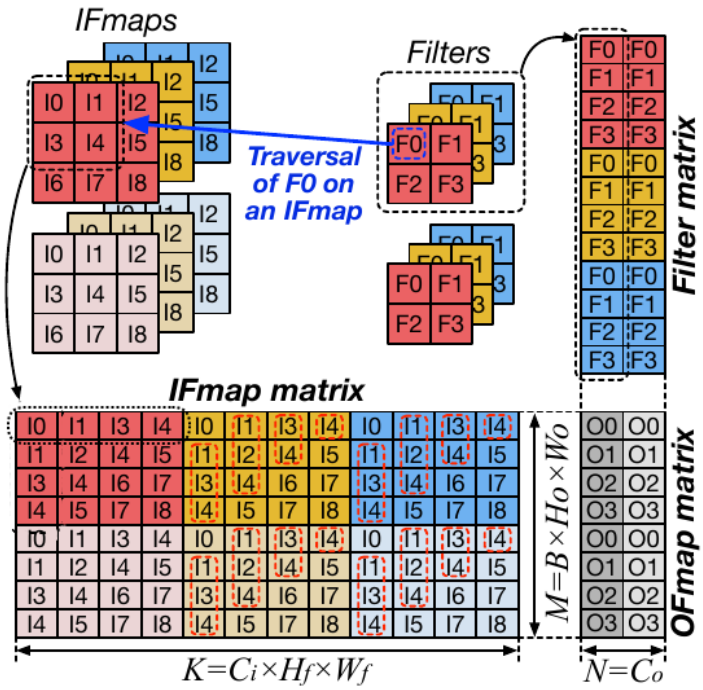


Figure 2.5: Im2col algorithm schematic [3]. ©2019 IEEE

are always connected to the internet, so cloud computing is the easiest and most common way to deploy a deep neural network.

The problem arises when it is needed to use these kinds of applications on devices that are not always connected to the web and that must be small and not consume much energy, i.e. on microcontrollers. Here the so called *edge computing* becomes necessary: the inference is run independently of the cloud and the internet, just relying on the device itself, which in most cases is not so powerful and it has strong storage, power and energy limits. This new field, called **TinyML**, has several advantages that are extremely useful in many applications and cannot be guaranteed by cloud inference and cloud computing [4]: reduce the network bandwidth needed to communicate with high performance devices by performing a first, high level investigation at the edge, being independent from network connection, which leads to better reliability; guarantee security and privacy, since the data collected by the sensors are only processed locally with no cloud transmission; reduce latency and guarantee real time response with no wait time for communication and processing on cloud; energy efficiency and low cost, more a must than an option on MCUs.

It is important to state and understand that in most cases the training process cannot happen on the device, since it requires a large amount of data and a long

time of execution. The common way in this case is to train the DL model on cloud, and then to perform inference on the edge device.

There are some ways to try to reduce the power consumption of neural networks [1], and most of them act on the number, the type and the way the operations are performed.

- A first way could be trying to reduce the actual connections between consecutive layers, in order to reduce the terms of the weighted sum. As it can be seen in the Figure 2.1, fully connected layers are the worst case here, since every single neuron has a weighted sum of all the previous layer neurons' output. The goal is therefore to limit the number of activations (neurons) that influence the following layer by making their weight (synapse) go to 0 by applying the proper non linearity or simply by **pruning** [4]. In this way a sparsely connected layer is obtained, which allows to use compression and as a consequence to use less memory and less energy (*windowed Neural Network*). However, the number of weights does not always reduce the energy consumption. With pruning, a threshold for the weights is set and all the weights below that threshold are canceled (set to zero). In this way the accuracy is reduced, but if the network with the new weights is retrained, most of the loss is recovered [4].
- Another interesting way could be the so called **weight sharing**: reducing the number of weights, to limit the storage requirements and see if there are some duplicates in the operations, in order to do them just once. This is because every activation has a different weight for every different destination of the following layer, so it could be possible to have just one weight for all the destinations [1].
- **Quantization** could also be a valid option to reduce the needed memory and to make the computational cost much lighter by reducing the size of the variables and use only simple types. An example could be limiting the number of bits of the results and using only integer numbers or fixed point numbers. This will of course come at the cost of a reduced precision, which most of the times can be allowed [4].
- **Knowledge Distillation** is used to transfer the knowledge learned by a powerful trained model (called *teacher*) to a much smaller model (called *student*), which can fit on a small MCU. This method increases accuracy if compared to the case where the small model is trained independently [4].
- **Encoding** is sometimes used both to achieve high efficiency in memory usage and to reduce the size of the model, without any accuracy loss. It is often used on the weights [4].

- While all the previous methods act mostly on the computational part of the network, great advantages in terms of energy efficiency can be gained by acting on the memory side and on the dataflow, trying to reduce the number of accesses, the size and the type of memory used, as well as sharing as much data as possible and treating carefully the intermediate results. Of course this is possible if there is a certain degree of freedom on the device's hardware.
- Again, if the hardware is customizable, it is possible to move part of the computation from the processor to the memory or to the sensor itself to reduce computational effort at the cost of reduced precision.

More strategies are even more effective, but they can be applied only once the type of used neural network has been decided, knowing in detail what operation has to be performed.

2.1.3 Neural Network Frameworks and TinyML

As it can be easily understood, creating and deploying a NN is quite complex, therefore some tools have been developed to make it easier and faster, as well as to create the possibility to have an open source environment where the whole community can improve the system. Such tools are called *frameworks* and they basically enable anyone who wants to go deeper in this field to start experimenting with common examples and building blocks to be used to create and train new DL models.

These frameworks support a wide range of cores and devices and provide libraries that allow for an easy and fast development of new DL models in many different programming languages, based on what is needed. There are also some transversal libraries and APIs that are available in more than one framework, such as Keras. Some of the most popular frameworks are Google's TensorFlow, Microsoft's CNTK, Facebook/Meta's PyTorch, Apache MXNet, Caffe and so on (Figure 2.6).

Figure 2.6: Most popular DL frameworks.

Each of them has its own advantages:

- TF has been the first one being developed, therefore it has the largest community and a quite extended documentation, as well as it ensures a great flexibility in terms of target implementation.

- CNTK is growing fast and it is known for being very fast in training and inference.
- PyTorch is becoming the standard in the research field, thanks to its ease-of-use, ease-of-debug and its unique approach of using dynamic computation graphs to represent the models, with a growing community that is already big enough to support any kind of user.
- Finally, MXNet is also growing fast, due to its great portability, its very intuitive and user-friendly interface (Gluon) and its approach of mixing imperative and symbolic paradigms for mathematical computation.

In the last years, the development of more and more frameworks created a compatibility issue, in the case it is needed to switch framework along the way for any reason. This is why a format called ONNX has been adopted by almost all the companies as standard accepted input format for the model, so that the design and training of a NN can be done with any framework of choice without any problem (PyTorch, TF, MxNet, etc.), and the model can later be saved in the ONNX format and converted into any other format to perform optimizations and inference.

In the field of MCUs, in particular, all the low power techniques that have been mentioned in Section 2.1.2 become necessary, since the devices have extremely limited memory availability and computational capability, and at the same time they rely on batteries that must last as long as possible.

In this case another problem is also the fragmentation of the MCUs, meaning their specifications can be very different from one to another, since they maximize performance with the available hardware, which is limited and always different. This is a new branch of ML that is growing very fast and is called **TinyML**. Various companies are investing in creating custom proprietary frameworks and compilers that support their MCU boards, such as Google (TF Lite Micro), Microsoft (ELL), Facebook/Meta (PyTorch Glow), ARM (CMSIS), STMicroelectronics (STM32 Cube.AI), NXP, etc. The fragmentation problem becomes even more important when the target device is based on a RISC-V core: even though there are several RISC-V standards to follow, the HW and the ISA can be customized in so many different ways that it is impossible to design a default NN implementation flow that works out of the box without asking the user to make important modifications to adapt it to the specific target.

However, the actual end-to-end frameworks are very few, in the sense that it is rare that the whole flow from the model design and training to the inference is managed by the same framework. Some examples of that are TF by Google (with TF Lite Micro) and CNTK by Microsoft (with ELL). In all the other cases there are lower level frameworks that take as an input the NN created by one of the main

frameworks for ML and what they do is optimizing and converting it to deploy it on some MCU boards. This is because some of the most popular frameworks such as Caffe do not support inference on MCU, even though they support edge-inference in more powerful devices like smartphones. Some examples of lower level frameworks are microTVM, CMSIS-NN and STM32Cube.AI, that accept input models coming from the above mentioned libraries. As it will be explained in Section 3.2, since the framework of choice has been TF (and more specifically TF Lite Micro), in the next Sections the focus will only be on this framework.

Interpreters vs compilers

When it comes to perform inference on MCUs, having a C or Python program is not enough anymore, but it is needed to go down to the machine code that can be executed on the available HW, since the device does not have any Operating System. As a consequence, new kinds of frameworks are needed in this case, and they can be divided in two main categories: *interpreters* and *compilers*. The most popular TinyML framework uses the interpreter approach (TF Lite Micro), while almost all the competitors have chosen the compiler approach (microTVM, PyTorch's Glow, ONNC, deepSea, etc.), since it reaches te best optimization possible in terms of HW and memory requirements and inference time. Some of them are built and targeted just for one brand, since they are developed by the producer of that brand (see STM32Cube.AI), while others try to be more generic and target multiple cores and boards, like the ones mentioned above.

The main difference between the two types is the fact that compilers need to go straight from the model to the machine code, therefore they need to be fully aware of the target's HW and perform very specific optimizations *a priori*, going through several intermediate steps and representations (see Figure 2.7, relative to microTVM). This can be done thanks to a frontend (that manages the model conversion from the ONNX format to the first IR), a middle-end (that handles the optimizations both on scheduling and on HW usage) and a backend (that takes care of the code generation by using the compiler backend LLVM in most cases, implementing the actual *fine-grained* HW operators).

On the other hand, the interpreter follows a different approach, just working at run-time selecting the proper operators to use and their optimized implementation, which still needs to be designed in advance. The inference in this case is simply reduced to an *invoke* call to the interpreter, which will handle both the memory usage and the operation flow on the run, in a transparent way. This makes the code extremely portable and flexible, with the drawback of a slower inference overall. More detail on this will be given in Section 2.1.3, since the chosen framework is TF Lite Micro, which adopts an interpreter approach.

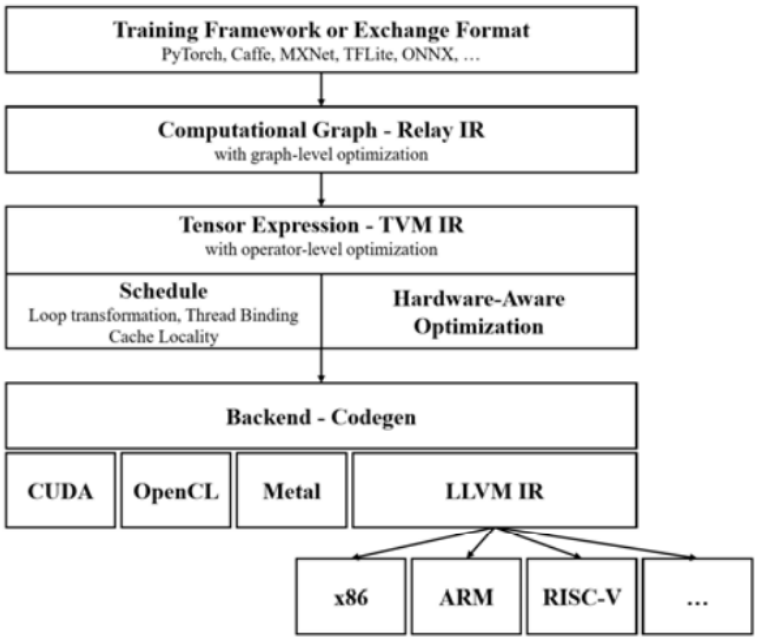


Figure 2.7: TVM optimization flow [5]. ©2020 IEEE

TensorFlow

TensorFlow is Google's framework and it is one of the most common ones. It is written in C++, even though it also has a Python interface. At the heart of TensorFlow there is the implementation of any algorithm by means of a *computational graph*, meaning that each operation in the algorithm will be a graph node, while the data that flows between operations is an edge connecting those operations. It is important to know that also constants and variables are seen as operations in TensorFlow, as they have no inputs but as output they have a constant or the current value of the variable respectively. In this way the visualization and computation of the algorithm results is very simple and effective, since to get the output of the whole algorithm it is only necessary to assign the inputs and go over (run) the graph backwards from the output to the input.

The data flowing between operations (i.e. the edges of the graph) is called Tensor(s) and they are symbolic handles to multidimensional group of values with an immutable type and size (called shape) [6]. As they are symbolic handles, they are just a pointer to where the values are actually stored, while the true values are stored through variables through some operations that give the initial values to them.

To create a neural network on TensorFlow, it is needed to make a *model*, and this is typically done by using *Protocol Buffers* format. "Protocol buffers provide a

language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way." [7]. In other words, they represent a data format that allows to define the data structures in a universal text file (.proto extension), being untied from any programming language. It is up to the protocol buffers tools to generate the actual code in the target language, which is later compiled and used as classes in the final code. In the case of TensorFlow, the graph that describes the algorithm is initially an object in the protocol buffers format, which will be translated into the target code and used as a class for instantiating in the code the object that represents the model (tf::Model in C++).

In practice some libraries such as the already mentioned Keras help and automatize the creation of a neural network, just by defining its structure already in the target language (indicating the number of layers and the number of neurons for each layer) and then compiling the model, which will be ready to be trained [8].

TensorFlow Lite

What was said above is generally valid for any kind of application that can be both trained and run on cloud, while in the case of edge computing, the inference must be run on the device, where all the computation must happen. To face the challenges of edge computing, Google has created a new version of the framework specifically thought for being lighter, that is why it is called TensorFlow Lite. This new framework has the goal of having a small code size (memory constraint) and of allowing a fast inference (computational constraint). At the base of this new framework there is a new model format for the neural network description: the *FlatBuffers* format. This format is still language-neutral and platform-neutral, but it is much more lightweight and avoids the necessity to deserialize the whole data before accessing it (needed with Protocol Buffers), which means faster access.

Also the neural network model in itself is described in a different way here, always with the operations (or operators) as central components that define the structure, but this time they are not organized as a graph, but they are in a put in a topologically sorted list, so that in order to perform an operation, it is just needed to loop on the list in order. In this way the structure does not require any preprocessing to satisfy the input dependences, but at the same time it is more abstract so there is the need for a few more lines to be executed at run time to get the information related to the actual implementation for the inference [9].

Another important difference with TensorFlow is that here the inference is interpreter based. This choice, even though is less efficient than the classic code generation flow in C, it is needed to increase portability and make it easier to modify the model. The interpreter is instantiated by giving it the model, the available memory and the operators needed to perform inference, and then it will handle the allocation and the initialization procedure for inference on the target

device at run time. In this way there is no need to recompile everything for each different target machine, since the target's specifications are not explicitly present in the binary machine code, but they are derived by the interpreter from a reserved memory area [9].

TensorFlow Lite Micro

In the case of inference on MCUs, TensorFlow Lite could be used, but in many applications it is not enough, since the MCUs are often much more limited than a common edge device such as a smartphone, both in terms of memory and of computational capabilities. Therefore, some more precautions must be taken and that is why a new branch of TensorFlow Lite has been developed, called TensorFlow Lite Micro. For this framework to be as general as possible, only basic operations can be assumed as available, and the quantity of memory used during inference (called *arena*) must be known a priori and reserved since the initialization, with no chances to increase it at run time.

The framework flow can be summarized in the following steps [9]:

- Create or convert the neural network model by using the FlatBuffers format. This can be done with the help of some API like Keras. The model can also be created in any other format and using any other framework, but it has to be converted into a .tflite file with FlatBuffers format by using the TF Lite converter.
- Create a suitable *arena*, which is an area of memory which is free and corresponds to the maximum memory that can be used during inference.
- Create the so called *OpResolver*, which will manage the implementation, the optimization and the final use of all the operators during inference. To limit the memory need and avoid waste, it is possible to generate and OpResolver that only contains the operators actually needed for the inference.
- Instantiate an interpreter, by giving it as parameters the model, the OpResolver and the tensor arena. The interpreter has three main jobs:
  - Load the model from the FlatBuffers format.
  - Manage the memory allocation by only using the *arena*. This is done very carefully, so that no memory is wasted, that is why there is a memory planner, which compacts the data stored in the memory, looks at the lifetime of every piece of data stored and allows memory reuse in case of non overlapping lifetimes. In addition it tries to reuse also the memory used by the interpreter during the initialization phase by deleting the unneeded data after that phase. Space for more than one model can be allocated, to have them stored together, but not run at the same time.

Finally, the user can personalize the usage of memory by adding some metadata in the FlatBuffers model description file.

– Manage the inference process when it is *invoked*, with the help of the OpResolver.

- The final phase is the execution, when the interpreter and the OpResolver will collaborate through an API to implement the operators, give them input values and run the model.

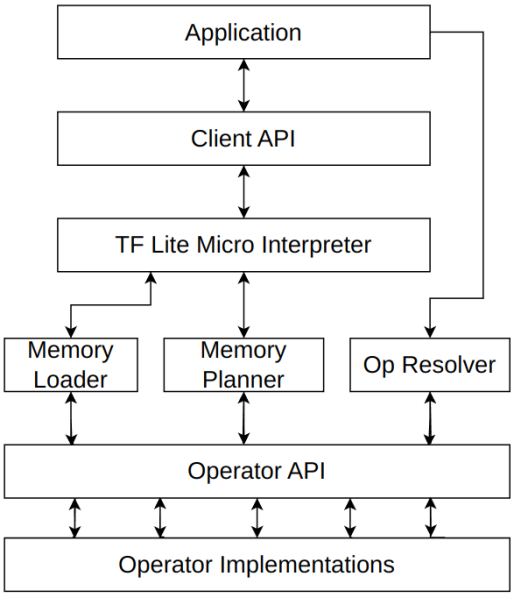


Figure 2.8: Typical framework flow in TensorFlow Lite Micro.

An important general consideration could be done here: the training process is always the same for all these kinds of framework (TF, TF Lite and TF Lite Micro), and it is done on cloud, since it requires high computational capabilities and a very high storage availability. What changes in practice between TF Lite and TF Micro is that in the Lite case the model is just translated into the lighter version and the program in high level language (C, C++, Python, etc.) can be run on the device during inference. On the other hand, in the Micro case, it is needed to have a byte array in machine language that can be understood and executed by the target MCU.

A final remark can be done about the OpResolver and the operators. Every operator corresponds to the so called *kernel*, which is the physical implementation of the operation, that could be a very simple operation (such as add or multiply,

also called *fine-grained* operators) or a complex one (such as a whole NN layer, having the so called *coarse-grained* operator). As it can be easily understood, the implementation is very dependent on how the core is made, so it is hard to generalize its implementation, and the optimizations must be done at the core level by the core producer. This is why the OpResolver often uses external libraries to implement the optimized operators. A classic example is for the ARM cores, since ARM itself created a library of optimized operators for its own cores called CMSIS. In this case in the compiling toolchain the CMSIS library is fetched and used when the operators are listed and assigned to the OpResolver.

2.2 RISC-V

During the last years, the RISC-V community has grown at a very high rate and an increasing number of companies has started investing on the development of RISC-V cores, which represent the open-source present and future of the MCUs. To understand what RISC-V is, it is useful to start from what an ISA is. A nice definition comes from the ARM website [10]: “The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done", and more in detail “The ISA defines the supported data types, the registers, how the hardware manages main memory, which instructions a microprocessor can execute, and the input/output model of multiple ISA implementations". Knowing that, there are two main kinds of ISA, being CISC and RISC. In simple words, in the case of CISC many more instructions are defined, even the most complex ones, that are rarely called, leading to the need of a large encoding space for all the instructions and a much more specialized HW that will take a lot of space on the chip. On the other hand, the RISC follows the opposite approach, having only the most simple and frequently used instructions implemented, while the complex ones will be executed as a sequence of simple ones, so that the HW is very simple, fast and optimized, allowing for smaller chips and faster clocks.

What differentiates RISC-V and ARM (that follows the RISC standard as well) is that RISC-V is open-source and it does not require to buy any license to be used and it can also be customized by adding new instructions or features, getting all the advantages of a growing community that develops it. In the following, some of the main RISC-V features will be detailed:

- There are 32 registers to be used (in the 32-bit version of the ISA), many more than the other ISAs (ARM has 16, x86 has 8).
- It has a modular design, composed by the so-called *extensions*. Each extension is a formed by a group of instructions related to one another for a particular

type of processing or computation, such as integer (**I** extension), multiply/divide (**M**), compressed (**C**), etc. Every extension is developed as a single batch by the community and when it has been tested enough it is released as stable, with the corresponding compiler support.

- The addresses of the input/output registers are always encoded in the same portion of the instruction, making the decoding very easy to implement and compliant with any possible addition or modification. Figure 2.9 shows the kinds of instruction of the RISC-V base integer 32-bit ISA.

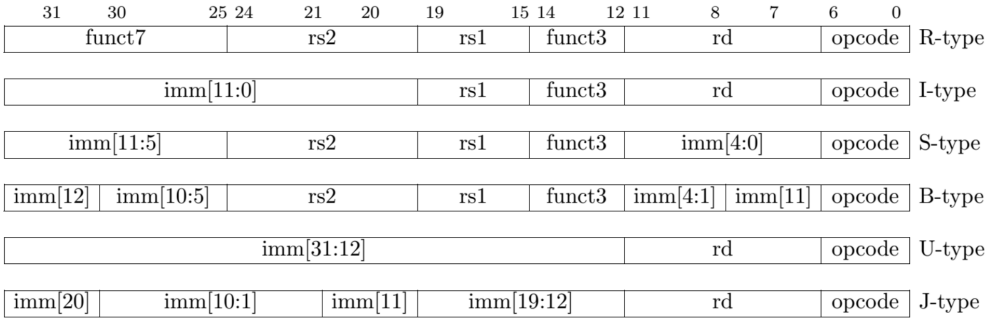


Figure 2.9: RISC-V Base Integer 32-bit ISA instruction types [11].

TinyML on RISC-V cores

Given the current trends in the research field, it is natural to think that the growing application field of TinyML can meet the growing HW design field of RISC-V to have a full open source edge ML application, on both SW and HW. The research is only starting going deeper in this aspect, but the interest is increasing and new research projects are being developed.

The fundamental point here is the opportunity to easily modify the RISC-V ISA with new instructions or new extensions that make the ML deployment faster and more efficient. In addition, RISC-V allows also to have complete control on the HW, therefore it is possible to create efficient instructions in parallel with the HW microarchitecture (i.e. accelerators) where these instructions can be executed in the best way in terms of power, speed, memory, etc. As it can be understood, the possibilities are quite promising, so the only thing left is for the research to create a proper infrastructure to make this whole process smooth and linear.

2.3 DL Applications in Literature

2.3.1 TensorFlow Lite Micro

TF Lite is quite known worldwide and TF Lite Micro can be considered as a mature framework, as it is growing very fast and more and more devices are supported. Here some basic examples are detailed, found in literature and made available in the TF repository, mainly to understand the common workflow that brings to the practical use of some neural networks with TF Lite on a MCU. The final choice of the application to be developed in this project will therefore be between the following examples.

Micro-speech

This is one of the most known examples with TF Lite Micro and the code for it is available in Github, ready to be compiled and flashed on several types of MCU boards. The goal is to recognize some particular keywords in a speech and it can be easily trained by using some available data sets for around 10 different words (such as “yes", “no", “right", “left", etc.). It has been optimized to work mainly with the ARM Cortex M- series cores and the neural network is composed by one 2D convolutional layer, followed by a fully connected layer with the Softmax activation function, that extracts the probabilities out of the fullly connected layer output. The convolutional layer accepts one input matrix (49x40, one channel) and has 8 filters (1x10, one channel each), therefore it outputs one matrix (25x20) with 8 channels (feature maps), which, after being flattened in a vector of 4000 elements, are given as an input to the fully connected layer. This layer will have 4 outputs, from which the softmax activation function will derive the output score of the whole NN (related to “yes", “no", silence and noise). Figures 2.10 and 2.11 visually describe the above mentioned network.

As stated in [12], this NN is not particularly accurate, but it has a very small footprint (20kB model using 10kB of RAM) and it could be used in an always-running low power device which, if necessary, will wake up a much more accurate system that performs a true speech recognition.

The particularity of this application is its input processing: in fact it does not work directly with the audio data from the microphone, but it works with spectrograms in the frequency domain as input. These spectrograms are 2D arrays where every row contains the FFT frequency results of a 30ms window from the audio sample data, after being averaged from 256 to 43 values. Every row is related to a 30ms-wide window, and every 20 ms a new window starts (so there are always 10ms of overlap). This is done for 49 times, and that is why in Figure 2.10 the input has 49 rows. Figure 2.12 provides a good summary of the input processing.

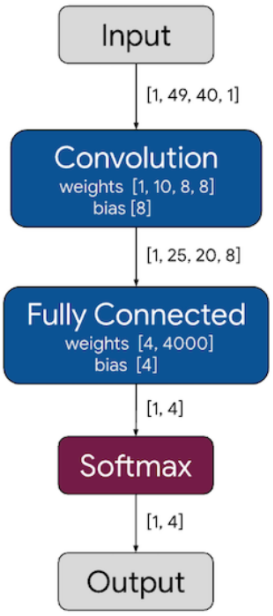


Figure 2.10: View of the NN model used for the micro-speech example.

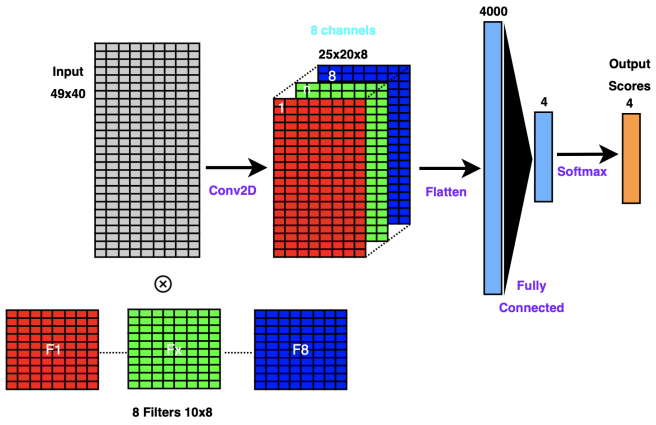


Figure 2.11: Micro-speech example: model graphic visualization.

The outcome is very satisfying (accuracy between 80% and 90%), provided that the input audio signal is “clean" enough, i.e. with a digital microphone the results are much more accurate than with an analog microphone.

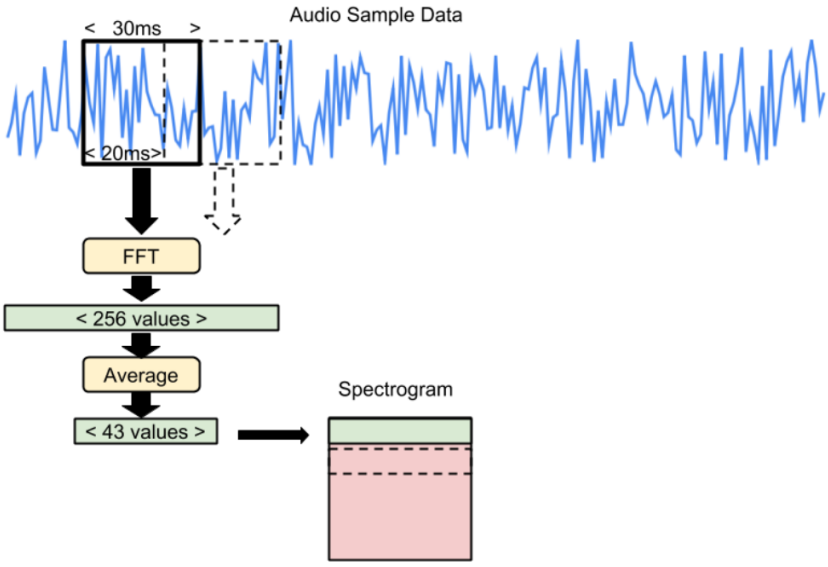


Figure 2.12: Input processing in the micro-speech example in TF Lite Micro [2].

Magic wand

This is another common example provided by TF Lite Micro, together with all the training data sets and code for several supported boards. The application uses the IMU to capture motion data and then it processes it to recognize a set of gestures (three in the basic example: wing, ring and slope). The model used here is a classic CNN model and the input processing is much easier if compared to the micro-speech example, since the model will accept the raw accelerometer output [2]. The model will take a total of 128 acceleration samples on each of the 3D axes, which, depending on the sensor's sampling rate, will correspond on a specific time interval analyzed at once. This data passes through 2 consecutive convolutional layers, each followed by a Max-pool layer to reduce the dimension. As a final stage, the data is flattened to enter a sequence of two fully connected layers (with ReLu activation function) followed by the usual SoftMax activation function to produce the output scores (probabilities) for each of the four learned gestures. Figure 2.13 gives a visual description of the NN model.

The inference will be performed many times per second, so each movement will be recognized multiple times by successive inferences. This allows for a better accuracy, if the true output is triggered only after the gesture has been recognized multiple times. Figure 2.14 shows the inference flow of this application, which corresponds to the structure of the C program to be compiled for the microcontroller.

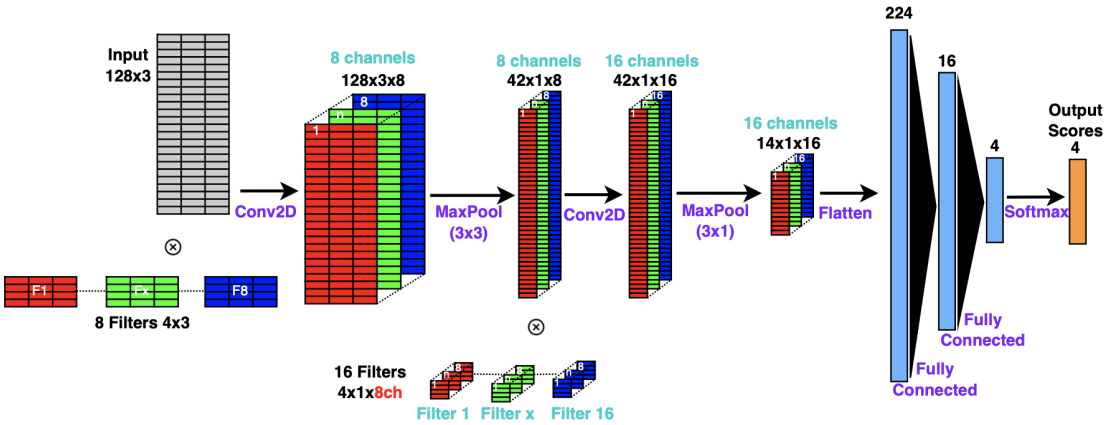


Figure 2.13: Magic wand example: NN model.

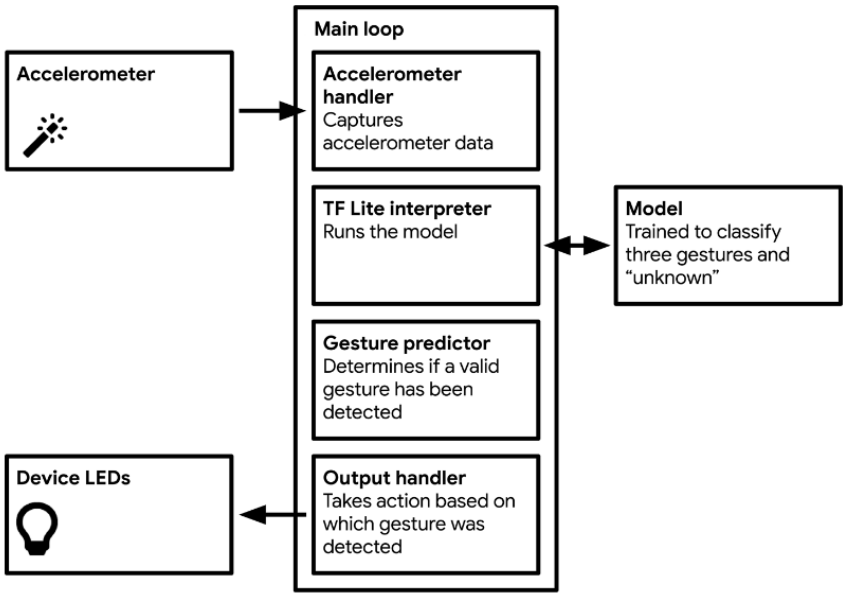


Figure 2.14: Magic wand example: inference flow [2].

It is important to say that in the example provided by Google, the model is not fully quantized to 8-bit integers, but instead it works with 32-bit floats both at the input, at the output and also for the network's biases and weights. The acceleration data has to be provided in the unit of milli-g (with 1g = $9.8m/s^2$, gravity acceleration), as 32-bit float numbers, therefore the pre-processing of the data is reduced to just a simple conversion operation from raw data given by the accelerometer's ADC to 32-bit float numbers representing the acceleration in

mg. Again the accuracy is fair considering the size of the model (20kB) and the computational capabilities of the MCUs. In [13], this application has been tested on the Arduino Nano 33 BLE Sense and the memory footprint has been measured, having a final result of around 80kB of flash needed for the application code and about 20kB of SRAM needed for the data, obtaining a final accuracy of 0.93 with an inference time of 55μs. With several optimizations, in [13] they are able to reach much better performance, but the optimizations are however purely focused on the model design, on the dataset and on the training, so they are out of the field of this project.

2.3.2 ML on RISC-V cores

In this Section some ML applications on RISC-V cores from the literature will be analyzed, focusing on the optimization possibilities that have been studied and tested both by SW and HW.

HW perspective

When creating a RISC-V ISA extension, the HW must support that, so the most common approach is to use an existing RISC-V core supporting just some of the basic ISA extensions (such as integer I and compressed C) and make it work with some custom developed accelerators (called AFUs). These AFUs map the new ISA extension or instructions, so that the main processor pipeline will only execute pre/post processing and control, while each AFU will implement (and accelerate) a specific part of the NN, according to the new instructions. However, it is important to understand that also from a SW point of view the toolchain has to be updated to support the new instructions (compiler, linker, assembler, etc.), as it can be seen from Figure 2.15.

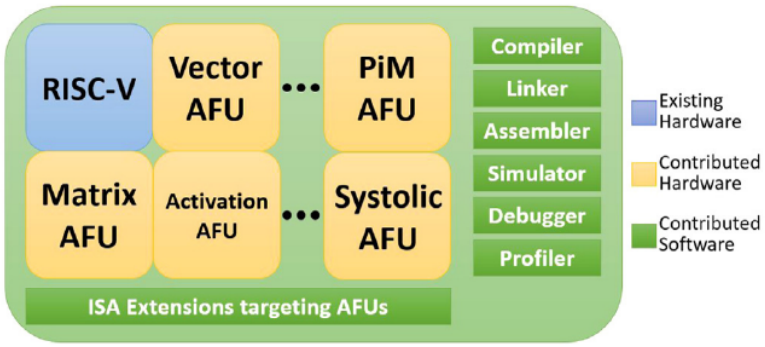


Figure 2.15: High level description of a RISC-V platform with custom AFUs [14].

As explained in [14], there are several ways to physically implement an HW accelerator, and they differ mostly on where exactly the accelerator is placed and how it communicates with the CPU (Figure 2.16).

- The most common way to do that is by having a standalone accelerator (in this case called APU, or coprocessor, top-left of Figure 2.16) that shares memory and resources with the CPU and communicates with it through a bus. This is not the most efficient way, since the overhead introduced by the continuous communication for exchanging data and control could drastically reduce the benefit coming from the execution on the accelerator [15]. A way to make this method worth it is to carefully design a way to efficiently reduce the memory accesses and the communication with the core. In [16] there is a very detailed example of coprocessor design for CNNs, with the whole microarchitecture steps and block schemes explained, with particular care to the data access.

- A different way could be to exploit the progress made on the memory architecture design to have a 3D RAM memory with the APU stacked on it (bottom-feft of Figure 2.16). In this case the data access is much faster and not a problem anymore for the APU, but the communication with the CPU still happens through a shared bus, so the control overhead remains. This kind of optimization is very effective if frequent and large exchanges of data are needed (typical for coarse-grained operations), which is not the case of interest here.

- Another approach could be to integrate the APU directly in the memory, as already anticipated in Section 2.1.2 (bottom-right of Figure 2.16). This kind of accelerator uses the analog properties of the memory to perform some of the typical NN operations through the simple access of the data in the memory. The most typical operation that can be performed in this way is the MAC. The problem here is that only a subset of operations can be accelerated with this method, so this approach is not suitable for all cases.

- The last method is to have the so called in-pipeline accelerator, meaning that it is integrated in the CPU and it will work exactly as the other functional units such as the ALU and the FPU (top-right of Figure 2.16). This is also why it is called *functional* unit (AFU) and not *processing* unit (APU). In this way the great advantage is that there is no communication overhead since the accelerator is already inside the CPU and is integrated in its pipeline, but at the same time this method will only allow to accelerate fine-grained operations and not entire NN layers. Another advantage is that several AFUs can be designed and integrated (see Figure 2.15), in order to accelerate more than one operation, of course depending on the available HW.

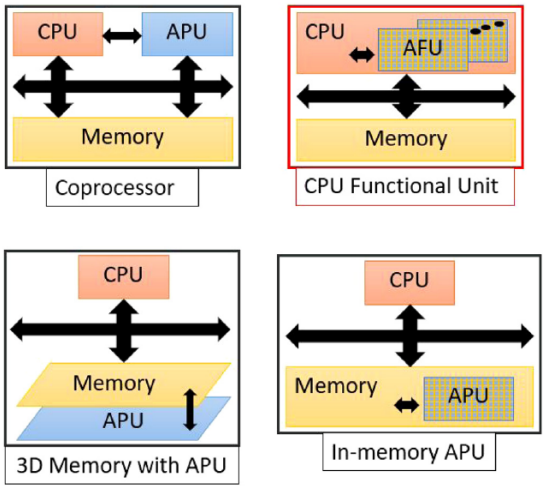


Figure 2.16: Different ways of physically implementing an accelerator [14].

As it can be easily understood, the most suitable approach for the implementation and acceleration of TinyML on a simple RISC-V core is the last one, because the NN are very simple and the data batches are never too large, given the strong memory constraints. A very clear and detailed scheme of how the in-pipeline AFU can be implemented is given in [17] (showed in Figure 2.17), where a simple 5-stage pipeline RISC-V processor is designed with 4 custom ISA instructions to perform vector operations (essential in CNN).

The researchers explicitly explain how they defined the microarchitecture to handle both general and custom instructions in the different pipeline stages. In particular the CNN accelerator has optimized performance thanks to a technique called *ping-pong operation* [17]. This is done to avoid long waiting time for the vector operands to be fetched and ready for the computation. The operands are not stored into special vector registers, but the VLOAD and VSTORE operations will act on a special buffer that is then connected to the external flash memory. The technique consists of having two different buffers and exploiting the data independence between feature graphs in CNNs to perform two operations in parallel (related to two different feature graphs), by alternating the used buffer to hide the latency of the memory access and keep the CNN AFU always busy. Figure 2.18 shows the instruction flow diagram of this technique.

A different example is described in [18], where there is no need for special buffers and they try to use only the RISC-V RV32IM base ISA with few custom instructions to manage the accelerated MAC operations. In this case the optimized operations are related to special kernels that perform three 8-bit MACs in parallel. This works only for a quantized CNN, where the general purpose registers (32 bits)

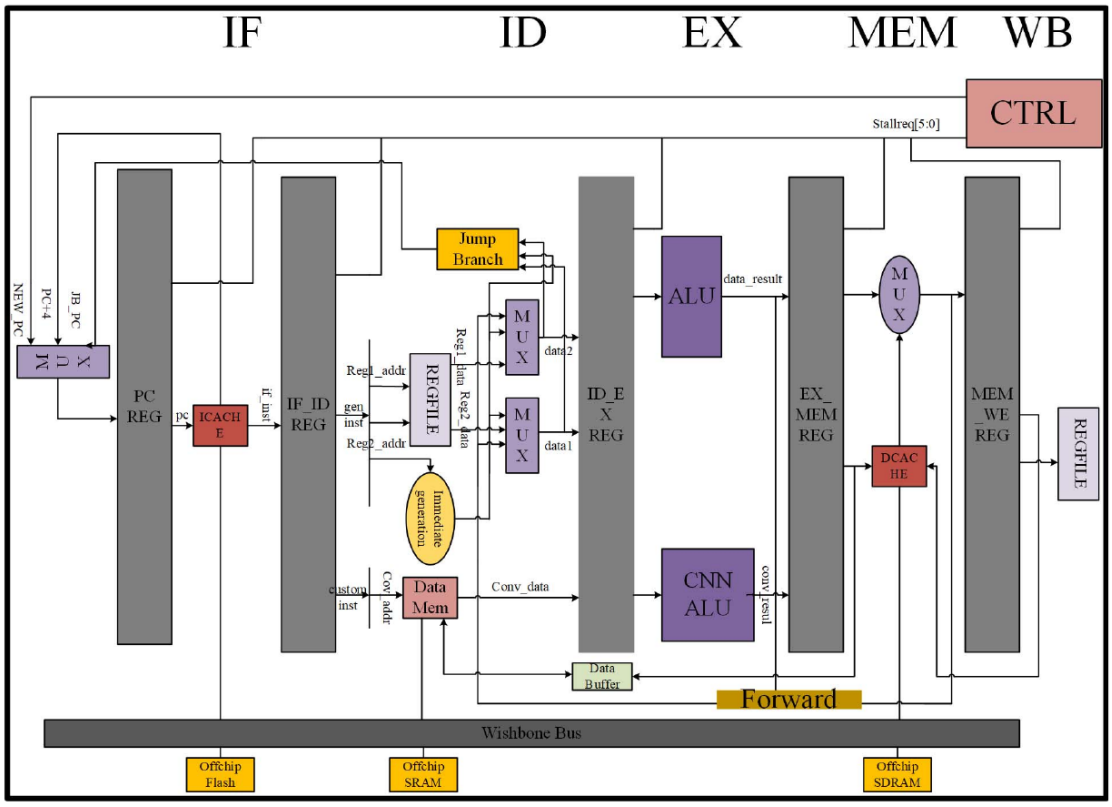


Figure 2.17: CPU structure with in-pipeline AFU [17]. ©2019 IEEE

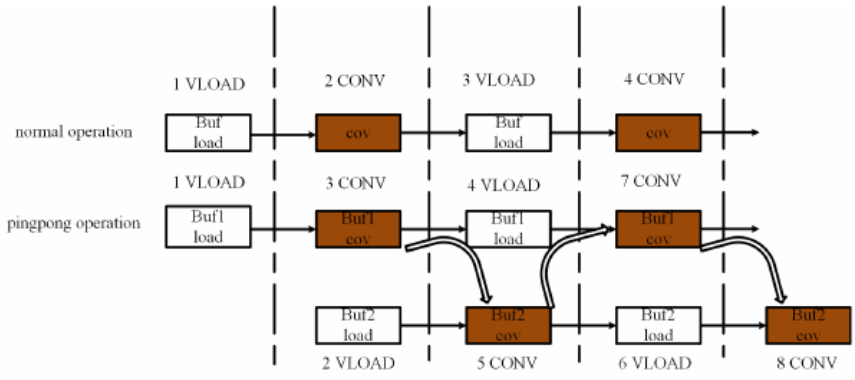


Figure 2.18: Instruction flow diagram for the ping-pong operation [17]. ©2019 IEEE

can be used as 4-element 8-bit vector registers with no HW changes. Since they want to optimize the specific case of 3x3 filter shape, to exploit the symmetry it is better to have each 32-bit register as 3 element vector register, with the final 8

bits padded, while the MAC HW block (called SIMD MAC) will have three 8-bit multipliers and 3 adders, like in Figure 2.19.

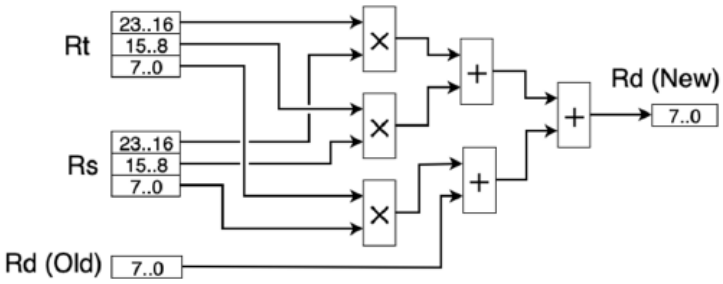


Figure 2.19: Block diagram for the 3x3 MAC accelerator [18]. ©2019 IEEE

SW perspective

From a SW point of view the idea is to create new RISC-V instructions that correspond to some basic and frequent operations that happen during NN inference, which should directly map to the designed AFUs. In most cases the operations will be very similar to vector and matrix operations, as well as MACs, therefore a good idea would be to start from the V (vector, SIMD) extension of the RISC-V ISA and take few instructions from there or modify some of them to fit better the application. It is not convenient to include the whole V extension, since it is quite bulky and it could not be supported by the basic RISC-V CPU that is available.

Another option could be exploiting the P extension, which supports the *subword* SIMD instructions, or better the **P**acked SIMD instructions. This extension is not as advanced as the V in terms of development and integration, but in the past few years the interest in it is growing, since it enables efficient and low power data processing, with 16-bit and 8-bit instructions intended for fixed-point or integer numbers. This descriptions seems to match exactly what are the needs of TinyML, with low precision and limited vector length. In [5] they have used the P extension instructions to implement the convolution computation during NN edge inference. They used the TVM framework and mapped (through tensorization) some TVM IR nodes to HW primitives that correspond to RISC-V P instructions in the LLVM backend. This process is well described in Figure 2.20, where it is also highlighted that before tensorization it is needed to actually change the data layout so that it matches the dot product in the TVM IR, which will be automatically replaced by the LLVM intrinsic function that maps to P instructions.

As a final remark, to perform edge inference on RISC-V by using TVM, it is still needed to have the so called DLR, which is a TVM runtime that is essential in bare-metal embedded devices (same as the backend described in Section 2.1.3).

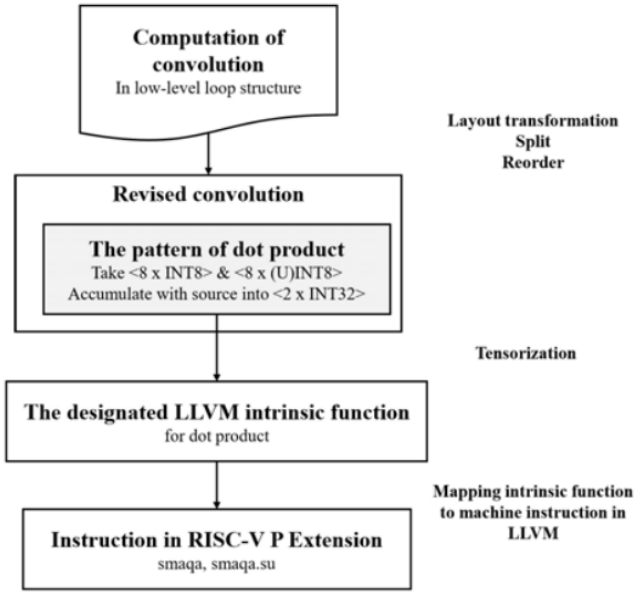


Figure 2.20: Mapping convolution computation to RISC-V P extension instructions [5]. ©2020 IEEE

This runtime will parse the model graph to get the operators, then it will load the weights and finally it will implement the operators. This is the final step to have an executable file that can run for example on a RISC-V simulator (Spike). Figure 2.21 shows the steps that bring to the final simulation.

From a general perspective, if there is the need of designing new custom RISC-V instructions, or include part of some already existent extensions, the process is different. The first thing to do is actually defining the instruction from an ISA perspective (opcode, number and type of registers, explicit operations that it describes, etc.). Once the instructions have been defined, it is needed to actually implement them and insert them in the flow that goes from the NN model design to the generation of the machine code ready to be flashed. In particular, this means to modify the compiler and the assembler so that they will be able to recognize and map the high level programming code (Python or C++) related to the accelerated operations to the new instructions. Two different ways to do that will be analyzed here.

One way is to act at a very low level, both modifying the C++ code and the RISC-V toolchain. This method has been adopted and explained in [15]. The starting point is always the design of the NN model by using a framework of choice and the following generation of the C++ code with all the general optimizations, by means of TF Lite, TVM or similar. At this point the C++ code must be modified

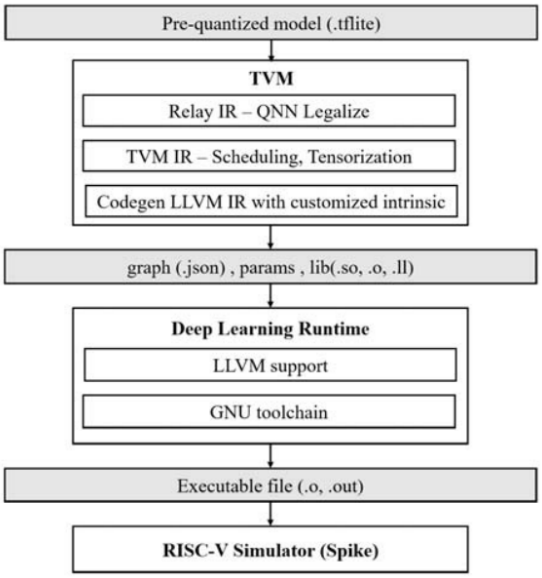


Figure 2.21: Overview from NN model to executable file through TVM compiler [5]. ©2020 IEEE

to add the high level macros that describe the new kernel operations and therefore correspond to the low level ISA instructions. This can be done through the so called *inline assembly*, where the new assembly instruction is directly specified in the C++ code. The next step consists of modifying the RISC-V toolchain (mostly the assembler) and building the proper runtime to realize the mapping to the kernels. The typical toolchain is the RISC-V GNU, and in particular the GCC and its assembler back end, the GAS. The inline assembly instructions are parsed and translated into machine code by the GAS, therefore the latter has to be modified to correctly understand and analyze the new instructions and produce the proper machine code for them. As a final step, the Spike ISA Simulator must be modified to support the new instructions and verify their correct functioning. The whole process is simplified on a scheme in Figure 2.22.

It is also possible to change the TF source code so that it supports a RISC-V target, by modifying the optimized operators that are target-specific and not portable.

Once all those steps are done, it is possible to cross-compile the C++ code for the RISC-V target, simulate and test it on the CPU.

The second way is proposed in [14] and it is called EXTREME-EDGE, that uses the so called HW/SW co-design methodology, which is explained in the following Section.

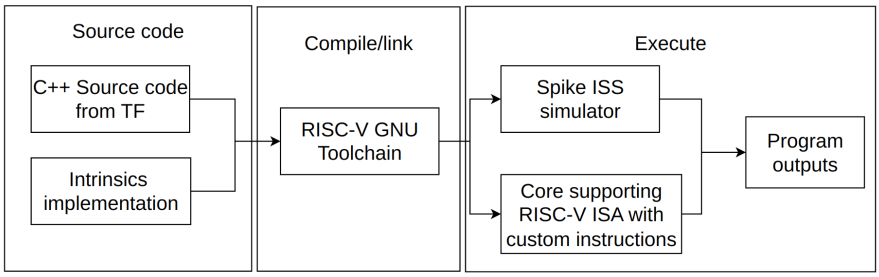


Figure 2.22: SW infrastructure using C inline assembly.

HW/SW co-design

This approach has the goal to make the entire design process of such accelerators very straightforward, fast and most importantly flexible and easy to update. In this way the progress made in the development of new instructions is immediately reflected on the microarchitecture design and vice-versa, and this is fundamental to stay updated with the continuous research evolution in terms of NN algorithms. This methodology also allows to avoid going deeper to the low level RISC-V toolchain, since modifying it could be inconvenient, because it requires a detailed knowledge of the whole architecture and it is prone to bug and errors.

The initial step is again the design of the NN model in any of the most known frameworks, and after that it is optimized and converted into C code by TVM compiler. TVM also has a compiler and assembler back end based on LLVM that can produce optimized machine code for some specific targets, but here this framework is just used for the model-to-C code conversion. So far there is no need to change any source code, so this part is as easy as in the deployment of any MCU. At this point another software is used, which will actually apply the co-design: Synopsys ASIP designer, which starting from the C code and from the processor model (meaning the ISA and microarchitecture) will allow to develop and debug in parallel both HW and SW, as well as providing a personalized full toolchain (or better SDK) to finally produce the machine code.

It is Synopsys that provides all the characteristics initially detailed, since it will generate compiler, assembler linker, debugger, instruction simulator and profiler, as well as a synthesizable RTL for the entire HW microarchitecture. All of these can be easily modified and are all automatically updated whenever the microarchitecture or the ISA changes, to have an easy and fast development flow. The full EXTREME-EDGE overview is detailed in Figure 2.23.

The Figure clearly explains the advantages of such a system, that requires no deep knowledge of the source codes and the back end, and makes it possible to personalize, design, test and debug in such an easy way that an iterative process

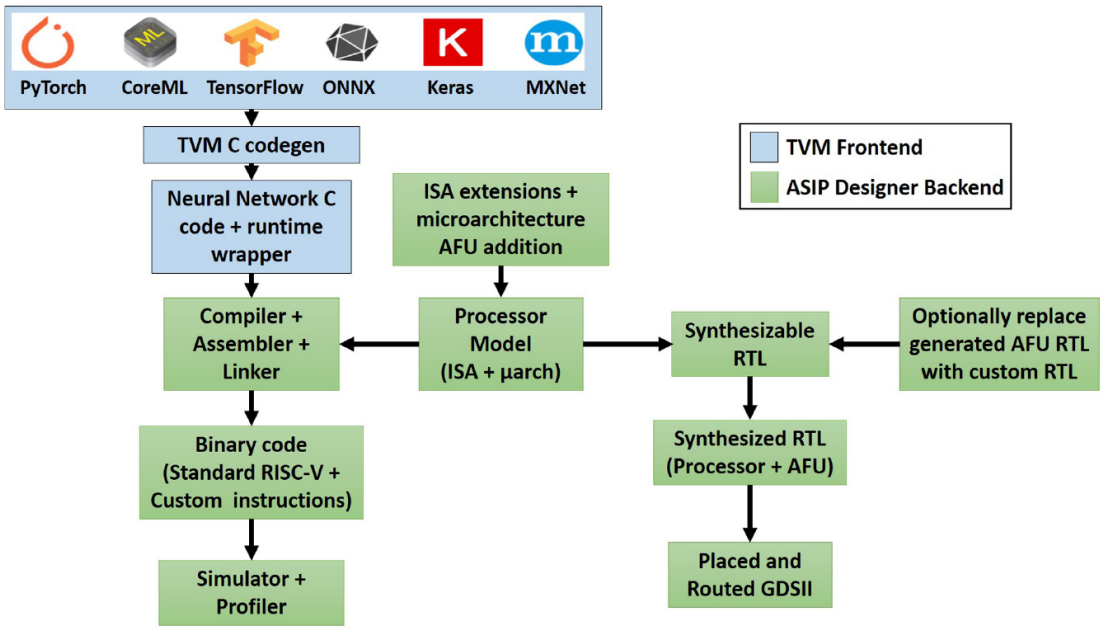


Figure 2.23: EXTREME-EDGE overview [14].

can be used to develop a project, without renouncing to the possibility to have a highly efficient implementation for the specific target.

2.4 Summary

The main consideration that can be made by reading this background Section is that the edge AI is a thriving research field in this moment, with many new opportunities coming up every month and new ideas being implemented to expand or optimize the possible applications. Much has been said, and yet much more still could be said about NNs, frameworks and possible HW implementation.

If a baseline is needed, it is reasonable to say that in this moment the standard for TinyML is still represented by TF Lite Micro applied to ARM cores, because this combination has been deeply tested, verified and optimized both by Google and ARM, introducing new features and support. At the same time, the research and market trends clearly show that in the near future things might change, going towards compiler approaches (Glow, microTVM, etc.) and towards RISC-V architectures, together with the always present need of being low-power.

Chapter 3

Methods

In this chapter it will be detailed the engineering process, the technical choices and the methodology that have been adopted to produce the final results, in such a way that any reader with a basic knowledge of the generic topics of this project will be able to obtain the same results and make his/her own optimizations and conclusions. The following Sections will go over the details of each step: Section 3.1 details the research process in general, Section 3.2 explains the options and the final choice of the most suitable AI framework adopted in this project, in Section 3.3 the choice of the reference application is analyzed, in Section 3.4 it can be found the name and model of the development board with ARM core used to have an easy start with the selected framework, as well as the exact framework version used in this project and the relative issues. Finally, in Section 3.5 the HW used in the project is analyzed in detail, including the measurement setup and related considerations.

3.1 Research Process

As already described in the first chapter (Section 1.5), an incremental approach has been followed during the project development, and in particular regarding the methods and the initial preparation few major steps can be listed. These steps are described in the following:

- General and detailed literature research to understand what is a NN, how to develop and deploy it and how nowadays it is implemented on microcontrollers.
- Choice of a test target microcontroller board commonly supported, needed to have a guided starting point.
- Choice of an AI framework that supports the selected target and eventual different implementations (i.e. RISC-V).

- Select a simple application among the popular ready-to-use ones that will constitute the SW core of the project.

- Prepare the HW setup both for testing and measuring.

As previously stated, this approach guarantees that the basics are well understood and applied little by little, so that it is much easier in the final stages to know where major and minor improvements can be made. The first step has already been deeply analyzed in the previous chapter, while the following Sections will try to explain better the other steps of the process.

3.2 Framework choice

As already stated in Section 1.6, the design and training of the network are not part of the thesis project, therefore the choice of the framework does not consider the most common characteristics of a framework (scalability, flexibility, speed, etc.), but it focuses more on what is specifically needed in this case, meaning the possibility to implement and deploy a NN on a microcontroller and the support that can be found on the internet for this particular application. As an example, even though PyTorch has become the research standard framework for its great usability, flexibility and community, its compiler for edge applications (Glow) is not developed enough and it does not guarantee the needed support for what is needed in this project. Knowing that, the options can be reduced to just two, since most of the other TinyML frameworks and compilers (such as DeepSea, ONNC, ELL etc.) are quite recent and raw, therefore they have too restrictive and limited use cases and almost no support on the web. The two options are TF Lite Micro and MicroTVM, which represent two opposite ways of looking at the TinyML world: on one side the interpreter-based approach of TF Lite Micro and on the other side the compiler approach of microTVM (more information about that in the background chapter, Section 2.1.3).

In this case the choice has been quite easy, since it is clear that TF Lite Micro has a much bigger community coming from more than 4 years of open source development. This means that TF is more or less well documented, it supports many different targets and it has several ready made working example to start experimenting from the beginning. On the other hand, even though it is growing fast, microTVM requires an additional effort only to get started with it, also for its compiler-based approach. As a result, to avoid losing time with too many framework-related issues, TF Lite Micro has been chosen as reference framework for this thesis project.

3.3 Application choice

Once the framework has been chosen, it is needed to understand what kind of practical application can be suitable for such a project. As said in Section 1.6, the focus is not on the NN itself, therefore as a starting point it can be chosen a ready-made example provided by the reference framework, in this case TF Lite Micro. Luckily, this framework has several examples for microcontroller application, so it is possible to choose the best fit for the present case, which is very particular: the RISC-V core has only I, M and C extensions, so the whole application must be implemented in a fully integer arithmetic from input processing to output handling; furthermore, the available memory is very limited and the power consumption should be as low as possible, but all these elements are part of the HW optimizations that this thesis is focusing on. At the same time, the application has to implement a meaningful NN, meaning that it needs to have at least 3 layers and it does not have to be too simple.

All these conditions lead to the choice of having an application that consists mostly, if not completely, on the NN alone, because there is not much memory, computational power or energy to be used other purposes, such as heavy input processing or energy-hungry sensors to gather data. Knowing that, the application choice can be done more easily.

Between the examples provided by TF Lite Micro, apart from the classic *Hello world*, there are a few interesting and very practical applications that can have several use-cases in everyday life, such as *micro speech*, *magic wand* and *person detection* (the first two are better explained and analyzed in Section 2.3.1). Based on what said above, the selection between them easily leads to the choice of *magic wand*, simply because in the case of *micro speech* most of the computational effort comes from the input processing, which consists on a fixed point FFT (not suitable for the RISC-V core used here), while the NN itself is just too simple; in the case of the *person detection* the camera will most probably consume too much energy, therefore its is not suitable for the purpose of this project. On the other hand, the *magic wand* seems perfect for this case, since it has almost no input processing, it uses a simple accelerometer to gather input data and the NN is simple but complete, with both convolutional and fully connected layers (see Section 2.3.1). The only potential problem in this application is that it is built based on floating point arithmetic by default, but this can be addressed as part of the thesis optimization work, as it will be explained in later Sections.

3.4 Board selection and framework version

As previously stated, the starting point of the experimental phase of this project consists on trying the chosen NN application on a commercial board equipped with an ARM core. In particular, the SparkFun Edge board was selected, since it is supported by TF and it has already a 3-axis accelerometer embedded.

The board has a low power working mode and it seems perfect for the purpose of this thesis, but unfortunately the last TF versions do not support the board anymore, therefore the newest stable TF version that still supports this board was selected, and this corresponds to **TF 2.6.0**, released in August 2021. This version is surely not the newest, but it has everything needed for this project (including RISC-V target support), and everything is included in the main TF repository, while in the newer versions TF Lite Micro has its own separate repository. It is also not possible to go too back in the past with the versions, both because the support for RISC-V becomes insufficient and because in the past the quantization was done by adding quantization and dequantization operators in the NN model, leaving the inputs and outputs to floating point and requiring floating point operations to quantize/dequantize, which is not possible with the RISC-V core in use here.

As a final remark, it is very important that the whole process is done within the same TF version, including eventual training and quantization, because if those steps are done with another version, several incompatibility issues will arise, even though some kind of backward compatibility is ensured by TF.

TensorFlow 2.6.0 - characteristics
The structure of this framework has changed a lot through the years, and it is important to know that the latest versions have two separate repositories for TF and TF Lite Micro, while in the version chosen for this project everything is in just one big repository, i.e. all the files that make TF work, such as python and C++ libraries, all the operators implementations, all the makefiles that simplify the implementation process and all the examples and support for the possible target devices. The organization in folders is very complex and it is not easy to understand it starting from the final make command that generates the binary ready to be flashed, this is because there is a hierarchical structure of makefiles calling and including each other and/or other files and libraries.

In order to have a general understanding of the process that leads to the generation of the binary, it is extremely useful to analyze and be aware of the makefile system by carefully reading some files, with the essential condition that all that follows is only valid for the version adopted in this project (2.6.0). The main makefile (*lite/micro/tools/make/Makefile*) contains the most important information and dependencies and it calls the general libraries that include everything

needed, for example the integer operators are included here from the internal TF implementation. This makefile receives the target device name from command line and it therefore includes a new makefile that is specifically related to the target itself.

The target-related makefiles are contained in *lite/micro/tools/make/targets* and they include other libraries and specific dependencies for the target. In the case of the SparkFun Edge the new makefile is *apollo3evb_makefile.inc* because the microcontroller of the board is the Apollo 3 from Ambiq, while in the case of a RISC-V core the new makefile is *mcu_riscv_makefile.inc*. This allows to add and compile also all the libraries with the peripheral drivers and microcontroller drivers, as well as indicating exactly what core architecture it is used and therefore what compiler toolchain is needed with all the specific option flags. As an example in the *apollo3evb_makefile.inc* file the compiler options are updated in order to have the best fit for the microcontroller and also some specific libraries are included (the so called BSP).

Important notice: the *mcu_riscv_makefile.inc* file that comes with the official stable repository of TF 2.6.0 contains a bug that makes impossible to use the riscv gcc toolchain. As a consequence, this file must be modified following what is written in [19].

A different makefile can be used to include some optimized implementations for the operators, just like it happens in the case of the CMSIS library. This particular case will be analyzed later in this Section.

One more important makefile is included and this is the one specifically related to the application and it can be found in the application folder, together with the code. It includes all the .cc, .c and .h files related to the application being implemented. An example is the makefile.inc file related to the *magic wand* example that can be found in *lite/micro/examples/magic_wand*.

To sum up, the main makefile will include at least 3 more files with the .inc extension: the target makefile, the application makefile and the optimizations makefile.

Another fundamental step needed to understand this framework is to become familiar with the operators system. The basics have already been explained in the background chapter (Section 2.1.3), while here it is explained how the operators are actually implemented and inserted in the repository. The operators implementation is called *kernel* and it is done in C, with some in-line assembly. There is both the builtin (TF Lite internal) implementation and the ARM optimized implementation from the CMSIS library. The first step when an operator is needed is creating its description in a C++ file. These files are located in the *lite/micro/kernels* folder and here there is also the CMSIS folder with the C++ files for ARM operators.

In this folder there is one .cc file for each operator, where the inputs and outputs types are checked and there is an *eval* function, which calls the real implementation

of the kernel. As a consequence, these C++ files are just a bridge between the program application and the kernel implementation.
The real implementations for the floating point operators can be found in *lite/kernels/internal/reference* in the case of the internal operators and each is defined in a .h file. There is a separate folder (*integer_ops*) for signed integer operators. All these files are included in the project by the main MakeFile (the one in *micro/tools/make*).

The CMSIS operators also have a .cc file as a bridge as above said, but their implementation is in *micro/tools/make/downloads/cmsis/CMSIS/NN/Source*, with one folder per each operator, which contains a different .c file for each possible implementation of that operator (the quantized ones are identified with the _s8 or _u8 suffix in the file name, meaning signed or unsigned 8 bit int). These files are included in the project only if the variable OPTIMIZED_KERNEL_DIR is set to *cmsis_nn* and passed to the main makefile through command line. In this way the cmsis.inc file mentioned above is included from micro/tools/make/ext_libs, and it will add all the optimized operator implementation source files. All the functions defined in these .c files are actually declared in just one .h file (*lite/micro/tools/make/downloads/cmsis/CMSIS/NN/Include/arm_nnfunctions.h*).

3.5 Experimental design and Planned Measurements

In order to get some quantitative results it is needed to perform some measurements on the system to show the progress made. In this particular case the performance and the efficiency of NN inference have to be measured, therefore the quantities to be measured are mainly four: the number of instructions executed during inference and in general of the application (which gives an idea of what is power consumption), the inference time, the model accuracy, and finally the size of the binary to be flashed on the target device, which includes the size of the NN model. Some of them (such as model accuracy and binary size) can be measured by software in advance, without any hardware test, while the others must be measured during the functioning of the system with the appropriate tools.

3.5.1 Test environment

As already explained in the previous Sections, the project has a preliminary phase based on the use of the SparkFun Edge board and a later experimental phase where the constrained RISC-V microcontroller is used in its FPGA implementation model. It is useful to get some measurement results for both the cases, so that a

comparison can be made in the end. Some quantities like the software-calculated model accuracy are independent from the hardware implementation, therefore they will be calculated just once, while the other values are expected to change based on the underlying hardware, so they will be calculated in both the contexts.

3.5.2 Utilized Hardware/Software

In this Section it can be found a detailed overview of the hardware and software used throughout the thesis development, with a separate Section related to the measurements setup.

There are two separate setups to be built, one concerning the SparkFun Edge board and the other one for the RISC-V FPGA model. In both cases it is needed to have the board itself, a programmer that downloads the application binary file into the flash (or RAM) memory, and finally the sensor to gather data, in this case a 3-axis accelerometer.

Regarding the **SparkFun** setup, the selected board, as already said, is the SparkFun Edge Development Board, in particular the first version (different from the Edge 2). It is equipped with the Ambiq Micro's latest Apollo3 Blue microcontroller, based on a 32-bit ARM Cortex-M4F processor that has a working frequency of up to 48 MHz, which can be increased (and in this application *will* be increased) to 96 MHz thanks to the TurboSPOT technology. The microcontroller has 1 MB of flash memory and 384 KB of SRAM memory, which are much more than sufficient for this application. This development board has also two embedded analog microphones and one 3-axis MEMS accelerometer ready to be used. However, the quality of the microphones is quite low, and this is one of the reasons why the *micro speech* example application was not implemented, since testing it would have become too complex. The accelerometer, on the other side, is the LIS2DH12 by STMicroelectronics, with 12-bit digital output and up to 1 mg/digit in high resolution mode at 2 g full scale. The output data rate can be between 1 Hz and 5.3 KHz and there is a FIFO buffer that can store up to 32 acceleration values for each axis. The interface can be both I2C or SPI, but it is connected to use the I2C protocol. This accelerometer works well, but cannot be used on the RISC-V setup, therefore the application on the SparkFun has also been implemented using a different accelerometer, compatible with the FPGA carrying the RISC-V core. It is the Pmod ACL by Digilent, which uses the Analog Devices 3-axis digital accelerometer called ADXL345. It is very similar to the LIS2DH12, with the same possible interfaces, same FIFO buffer and same full scale options. The difference is that this one has a 10-bit output with a maximum resolution of 4 mg/digit at 2 g full scale and also the output data rate can go from 0.1 Hz to 3.2 kHz. As a final remark, there is a small detail that could make the difference when testing:

the outputs of the two accelerometers can be collected in a 16-bit integer variable, but the LIS2DH12 provides *left-aligned* data, while the ADXL345 provides it *right aligned*. Table 3.1 summarizes all the details just described.

Accelerometer	**ST LIS2HD12**	**ADXL345**
Full Scale	±2 g/±4 g/±8 g/±16 g	±2 g/±4 g/±8 g/±16 g
Output interface	I2C/SPI	I2C/SPI
Output data rate	1 Hz to 5.3 kHz	0.1 Hz to 3.2 kHz
High-resolution mode	12-bit @ ±2 g full scale (1 mg/digit)	10-bit @ ±2 g full scale (4 mg/digit)
Output data alignment	Left aligned	Selectable (right alignment selected)

Table 3.1: Accelerometer specifications - comparison.

The Edge board can be programmed with a signed binary file by using the SparkFun Serial Basic Breakout and a simple USB cable, thanks to the SparkFun Variable Bootloader, but in this project a different programmer has been used by accessing the JTAG connector on the back of the board. In this way the raw un-signed binary is used and it can be downloaded directly to the board. The chosen programmer is the SEGGER J-Link EDU Mini V1, which allows to program the board through the J-Link Software and Documentation pack (in particular JLinkExe to program and JLinkSWOViewer to show the print messges from the application code). The EDU Mini specifically supports the Apollo3 Blue microcontroller under the code name AMA3B1KK-KBR and the interface supported by the microcontroller is the SWD, with a communication speed of 2000 KHz. Before using this programmer it is important to notice two details: first of all, to program the microcontroller by downloading a binary, the starting address of the memory where the binary will be stored has to be given by the user, and the correct location is 0xC000, which is the address where the first bootloader (the Ambiq Secure Boot Loader (ASB) by Ambiq) ends and the second bootloader (SparkFun Variable Bootloader (SVL) by SparkFun) begins. In any case the addresses before 0xC000 are protected and cannot be overwritten. The second important note regards the fact that to be able to see the printed text on the SWOViewer the option *printing over ITM* must be enabled by using the HAL function *am_bsp_itm_printf_enable* at the very beginning of the application C code.

The **RISC-V core** test setup consists of the Digilent Arty S7 board, equipped with a Xilinx Spartan-7 FPGA, which implements the core itself. The synthesis and programming are performed by the software Xilinx Vivado, in particular the version 2019.1, by using a specific script, while the simulations are run using Icarus

Verilog (updated to 2021-03-09) and the waveforms produced are visualized and analyzed by the software GTKWave, version 3.3.101).

The RISC-V gcc toolchain used to program the microcontroller is automatically downloaded and used by the TF makefile and its characteristics are given by the *PLATFORM_FLAGS* listed in the RISC-V makefile.inc (*mcu_riscv_makefile.inc* in *micro/tools/make/targets*).

The Arty S7 board supports the so called Pmod interface and has several Pmod connectors, which is why the above mentioned Pmod ACL has been chosen as accelerometer to be used in the final implementation.

The Spartan-7 FPGA is programmed with a basic test version of the constrained microcontroller, which includes the RISC-V core and few essential peripheral IPs (see Figure 3.1), such as the JTAG interface, the UART interface, the SPI and I2C interfaces, one 64-bit timer and a few GPIO pins. It has 64 kB of data SRAM and 64 kB (or 12 kB) of code SRAM, no flash memory and the clock frequency is 12 MHz. The core itself implements a 32-bit RISC-V ISA including the Interger (I), Multiply (M) and Compressed (C) instruction expansions, so there is no floating point unit, as already said in the previous chapters.

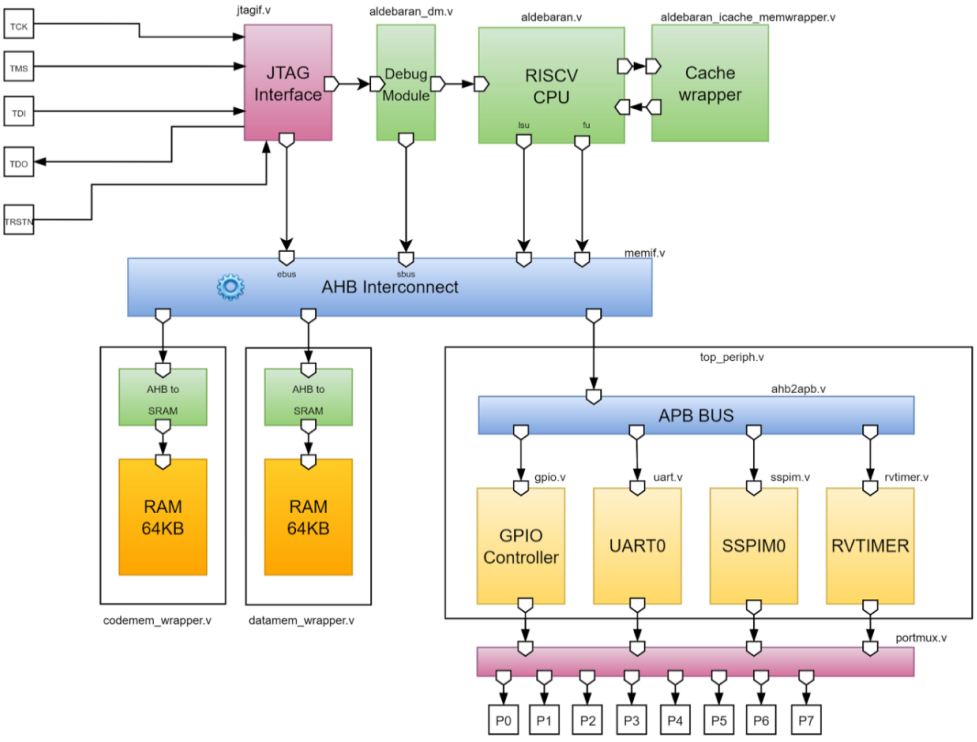


Figure 3.1: Simple scheme of the test implementation for the RISC-V microcontroller.

The microcontroller is programmed through the JTAG interface and the programmed used in this case is the Adafruit FT232H Breakout board accessed by a python script that takes the binary and downloads it into the microcontroller.

Measurement Hardware/Software

In this Section the setup needed for each of the quantities to be measured will be detailed.

The **accuracy** measurement, as above said, is done by software once the NN model is designed and trained, therefore it is closer to an ideal accuracy. It is ideal also because it is related to the model on its own, separated from the application or from the hardware and software that make the final application work. To better understand what comes next, it is convenient to first go through the steps that characterize the training of a network. In particular, at the base of training there is a large amount of data samples, called *dataset*, which must be as much diverse as possible, coming from different conditions and different users. In the case of this project the training that has been adopted in the TF magic wand example is called *supervised training*, meaning that the information contained in the dataset is labelled, so it contains in itself the information about what it is. To have a more practical example, the magic wand training data consists on accelerometer output data in the three axes, grouped batches of 128 acceleration values on each of the three axes, so that each batch represents a gesture that is analyzed during a single inference run. The final dataset is therefore composed by around 1000 different gestures, collected by 10 different people and conveniently augmented to mimic heavier noise of typical real cases. Having this data labelled means that each gesture batch has a label saying what kind of gesture it is, making it much easier to test and verify.

What is commonly called *training* is actually process split in three phases, namely training itself, validation and test, executed in this order and with three different datasets (see Figure 3.2). In a few words, during training the weights of the network are tuned so that the it learns eventual patterns or structures in the data, while validation is performed to understand in real time the effect of the weight change on the result and compare between several network structures. Finally, the test phase is done on the final trained network to see its performance. As it can be easily understood, to have meaningful results it is essential that these three phases are done with different datasets that are unbiased and independent from each other, therefore in the magic wand case the three datasets are derived from the main one by splitting it in 3 non-overlapping subgroups, each of which contains data from all the 10 people. A common ratio in this case would be having

60% of the total data in the training dataset, 20% in the validation and 20% in the test. In this way three unbiased, unique and independent datasets are created.

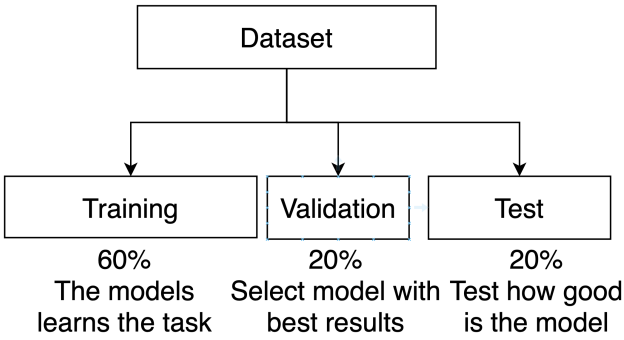


Figure 3.2: Training process: phases.

At this point it is clear that the model accuracy can only be calculated during the test phase and using the test dataset, and once the model has been trained and fixed, it remains the same independently from any application implementation. In the case of this project, it is interesting to understand and measure the loss in the ideal accuracy going through the optimization steps, being very careful to use the exact same test dataset for all the different network models so that the comparison is meaningful. Since the training, validation and test are performed only on the initial non-optimized network model, to get the accuracy for the .tflite and the quantized model it is necessary to perform inference by giving one by one the test gesture data as input, record the results and compare them with the labels that state the correct outcomes.

The **inference time** has to be measured on the target device during the application test, and it is important to know that this time interval does not include the input processing and the output exposure on screen or through LEDs, but it is the time needed by the model itself to get an output starting from the input given in the correct format. As a result, when running the entire application it is not easy from the outside to understand when this interval really starts, therefore the only way of measuring it properly is to use some tools internal to the device and perform the measurement inside the application code. This is why this measurement can be easily done by using the timer embedded in the microcontroller, which can be enabled right before the inference starts (after input processing) and disabled right after it ends (before the output processing).

In the case of the SparkFun Edge, the timer used is the so called *CTimer* (32-bit), which runs at 12 MHz and is enabled as a benchmark by default with some default functions to be easily used pre- and post-inference. In the case of

the RISC-V FPGA model, the only available timer is used as a benchmark: it is a 64-bit timer that runs at 12 MHz and the calculation of time elapsed is done by accessing the count register value before and after inference.

There is not much to say about the **size of the binary file**, since it is measured by software whenever the binary is produced, while the model itself has a size that shows the goodness of the optimizations when performing quantization, and this is given by the TF Lite converter. This reduced size allows to limit the quantity of data RAM needed by reducing the *tensor arena* size.

The **number of instructions** executed during inference can be measured by the microcontroller just like it in the case of the inference time, meaning that the measurement is run by the C program and uses the HW embedded in the core. In this particular case, it is done by using the so called CSRs, which are special registers included in the RISC-V Integer standard implementation. The CSRs are different from the general purpose registers and they have their own 12-bit address space (even though they are not that many), so that the user can add some more. Each of them has some specific functionalities, connected to a particular RISC-V instruction. In this case two of them will be used: the *mcycle* and the *minstret*, which are two 64-bit registers related to two corresponding 64-bit counters. *mcycle* contains the number of clock cycles executed by the core, while *minstret* contains the number of instructions executed (retired) by the core. These counters are automatically managed by the core and the registers can be reset by SW to the 0 value in order to have a starting point for the measurement. As a result, the exact same procedure used for measuring the inference time can be applied here, just resetting the counters right before the inference starts and then reading the counters' values right after the inference ends. The result is the number of instructions and the number of cycles required by one single inference run, from which also the CPI can be calculated, as well as the inference time itself, useful to check what has been obtained with the other method.

In the ARM core the number of instructions is calculated almost in the same way, but the registers in use are part of the so called DWT unit of the Cortex M4 core, which contains several counter to have the number of cycles spent by the CPU doing any of the main actions (multi-cycle instructions, exception, sleeping, loading/storing, etc.). Following what is written in [20], it is possible to calculate the number of cycles and the number of instructions in the exact same way as with the CSRs in RISC-V.

As a final remark, any other generic measurement mainly regarding output signals and communication between the microcontroller and the external peripherals is done by using a Digilent Digital Discovery, which is a device that can work

both as a logic analyzer, protocol analyzer and as a pattern generator. In the case of this project it has been used mostly to test the communication between the microcontroller and the accelerometer, i.e. SPI communication, as well as to perform function profiling and to measure inference time externally through GPIO pins toggle, in order to double check the timer measurement. To get the most out of the Digital Discovery it is best to use the Digilent software Waveforms, which makes the test setup and visualization very simple.

3.6 Assessing reliability and validity of the data and the methods

In the following Section all the measurements above detailed will be briefly analyzed from the reliability point of view, highlighting the precautions taken to make sure that the results are valid.

The accuracy measurement, as already broadly explained, can be seen as reliable, always taking into account that it is more like an ideal accuracy, since the model is run by software with some inputs that are ready to be used. The measurement is valid because the test dataset is not used during training, therefore the data given to the model is completely new for the model itself, and it is also unbiased, since it comes from gestures performed by different people. The main thing that makes the comparison meaningful is the fact that the test dataset used for all the measurements is always the same and the data needed as an input by the model is always the raw data from the accelerometer, eventually casted to 8-bit integer, so there is almost no input processing to be done on the test dataset. It is also worth it to mention that the used dataset has been obtained using the SparkFun Edge board, so the accuracy measurement for that board are perfectly realistic, while in the case of the FPGA model, the accelerometer in use is different, but it has been verified that the output data are very close to the ones from the SparkFun Edge embedded accelerometer (same full scale value, similar resolution).

Regarding the inference time measurement, it makes sense to say that the measurement itself will be very accurate, given the fact that both timers run at a frequency of 12MHz, so the uncertainty will be of the order of μs or tenths of μs, while the unit of interest here is the ms. This method allows also to repeat the measurement for each inference, so the results coming from several runs can be easily averaged out. Repeating this kind of measurement on a different device will of course require to change the code and use the specific timer available on the new device, but the whole procedure remains the same. Moreover, the fact that both the timers on the two boards run at the same frequency makes eventual

comparisons valid, since the uncertanties will be the same in both cases. One more thing that confirms the validity of the time measurements is that here they will performed in three different ways: using the timers, using the Digital Discovery with GPIO toggling and calculating the number of cycles through DWT or CSR. If all the three methods give the same result, then the measurement can be considered valid and accurate.

Another detail to be highlighted is that the inference time will strongly depend on the target's clock frequency, therefore it is hard to make comparisons between different target devices that have different clock frequencies. As a result, comparisons are meaningful only when changing or optimizing the software or hardware within the same target device, while in general given the inference time on a device it is possible to state if it is enough to allow a practical use-case to be successful addressed by the application.

Chapter 4

Implementation

In this chapter the implementation itself will be analyzed, going over all the details that brought to the final results. As already said multiple times in the previous Sections and chapters, the implementation has been split in separate gradual steps, first quite guided to get familiar with the framework and the application, and later more and more independent from the initial ready-made example to match the needs of the project and obtain some results. The macro steps can be summarized as follows:

- Implement the basic *magic wand* example from the TF Lite Micro (version 2.6.0) on the SparkFun Edge board with no modifications, i.e. with 32-bit floating point arithmetic.

- Modify (quantize) the NN model to have only 8-bit integer arithmetic, change the application code to match the quantized model, implement it and test it on the SparkFun Edge board.

- Change the accelerometer used by the modified (quantized) application from the LIS2DH12 embedded in the SparkFun Edge to the Pmod ACL (ADXL345), modify the connections and the application code, implement it and test it on the SparkFun Edge board.

- Convert the application to match a constrained RISC-V with the Pmod ACL accelerometer, i.e. switch compiler toolchain, modify makefile and application code, implement it and test it on the FPGA model of the microcontroller.

- Perform measurement on the previous implementation and find optimization spots where a better efficiency can be obtained, optimize the application, implement it and test it on the FPGA model of the RISC-V core.

4.1 Basic implementation on SparkFun Edge

The first step is fairly simple, but it may require some time to make the simple ready-made example work on the SparkFun Edge, because to really understand the mechanisms behind it it is needed to go through all the complexity of the TF repository, including its versions and the makefile system to target many different devices, all things already explained in Section 3.4. Moreover, it is needed to get some preliminary practice with the SparkFun Edge board, including how to program it and also how to read the code and its BSP library that is used to access the peripherals through HAL functions, as already highlighted in Sections 3.5.2 and 3.4.

Apart from what just mentioned, this first step is quite easy to complete with no issues. The flow that describes the application code can be seen as a scheme in Figure 4.1, where the average is done on 8 consecutive inferences in the default case.

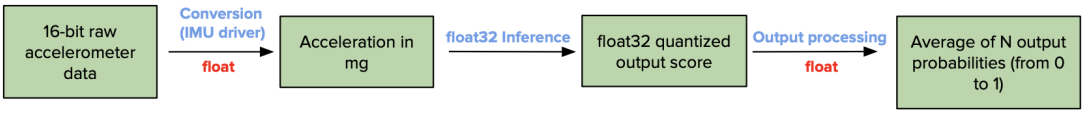


Figure 4.1: *Magic wand*, application code flow scheme.

A much more detailed dataflow for the application will be described in the next Section, since the quantized dataflow is actually the main focus of the project. However, it is already possible to have a general understanding of how the application works and how the code is organized. This has been explained quite well in [2], therefore here just a general overview is given. The code tries to mimic the Arduino structure, so the general main function calls a setup function at the very beginning and then it calls a loop function inside an infinite loop. These two functions are defined in the file *main_functions.cc*, together with some global variables and buffers needed by TF, such as the *tensor arena*, where all the variables handled by the interpreter will be stored. The setup manages all the initialization operations required by the system and by the application, importing and checking the NN, instantiating only the needed operators, setting up the accelerometer and retrieving some of the essential model parameters.

On the other hand, the loop function handles everything that needs to be done continuously, such as collecting and processing input data, running inference, processing the output scores and eventually showing the results through visual messages or LEDs. In this loop function some important measurements can be done with few more lines of code.

The management of the various functionalities and actions is spread between several .cc and .h files, together of course with all the BSP files and functions, as well as everything that comes from TF that controls the overall system. The specific application file were already briefly introduced by Figure 2.14, and are highlighted more in detail in the following:

- *accelerometer_handler.cc* (or *pmod_acc_handler.cc* in the version with the ADXL345 accelerometer): in this file everything related to the accelerometer is managed, i.e. the communication protocol and IP, the initialization, the reading and processing of the acceleration values, as well as the management of the big input buffer where the model input data is fetched from. The two main functions here are *SetupAccelerometer* and *ReadAccelerometer*. This file changes almost completely when the hardware is changed (meaning either the accelerometer and/or the whole target/microcontroller are changed).

- *gesture_predictor.cc*: this files controls the output processing, taking the output scores from the model inference and analyzing them to see if a gesture has been recognized. This is not trivial, since in the original application an average between several successive inferences is made. In this file there is only one function, *PredictGesture*, and this is generic for any implementation.

- *output_handler*: here what is handled is not the output of the model, but the output of the entire application, meaning how the result of the gesture_predictor is shown to the world. In particular based on the prediction result, in the function *HandleOutput* a LED is turned on and/or a text message is printed on screen to notify the user about the outcome. Again, this file changes depending on the HW where the application is implemented.

- *constants.h* : this header file is very important when it comes to optimize how the application works in the final implementation. This is because here all the important parameters are declared and initialized. In particular the average length and the threshold values for the output scores to recognize a gesture are defined and can be tuned by changing their value.

4.2 Model quantization and new implementation on SparkFun Edge

This step is crucial for having a true portable application that can be used by almost any target device. In particular, focusing on the ultra low power devices like the RISC-V core used in this project, it is very likely that the microcontrollers will not have any FPU, so floating point arithmetic is not an option. The original *magic wand* application proposed by TF uses floating point data and operations

from the very beginning to the output processing, as well as the model itself is made of floating point operators. Another problem could be represented by the number of bits needed by each variable: the default case uses 32-bit data, which is quite heavy for a constrained core with very little data memory. This is why in this project the first optimization choice has been made considering these two aspects and following the most common approach, already described in Section 2.1.2: quantization, meaning passing from 32-bit floating point to 8-bit signed integer arithmetic and data.

To make a full system quantization it is needed to act on two sides: the model and the application code. The model quantization is handled by the TF library in Python, while the application code must be converted manually changing all the variables and operations involved in the input and output processing.

4.2.1 Model quantization

To quantize the model it is needed to change the python script that trains it, without actually changing any parts of the training itself. This is because the kind of quantization method adopted in this project is called *post-training quantization*, which is usually opposed to the *quantization-aware training*. The latter is more complex to perform, but gives better results in term of model accuracy after quantization, because it simulates the behavior of the quantized model already during training, adjusting and tuning the weights accordingly to reduce the loss and increase accuracy. On the other hand, in post-training quantization the training process remains exactly the same and the TF Lite converter acts on the trained model changing all the weights and the operators from 32-bit float to 8-bit signed integer (fully integer quantization). As a result, the quantization step in this case is performed while converting the NN model from keras to TF Lite, and it is only needed to give some instructions to the converter by setting some parameters. In order to do that, it is of course necessary to have a fully integer implementation of the needed operators, and also the converter needs some preliminary information to perform a good quantization minimizing the accuracy loss. In particular, a *representative dataset* must be given to the converter, which is a quite large random sample of the typical input data that the model will receive, and this means creating a python function that yields input values fetched from the training dataset in a random way. Other parameters to be set before running the TF Lite conversion (and quantization in this case) are the input and output types (signed 8-bit integer in this case) and the supported operators (*TFLITE_BUILTINS_INT8*). In this way a new .tflite file is created, containing the quantized model (the app *Netron* can be used to verify the result and obtain the quantization parameters: slope and offset).

It is important here to pay attention to which version of TF is installed as a python library. This is because downloading the TF2.6.0 repo on the computer does not mean that the installed python TF library is the version 2.6.0, but python will use the version installed through the command *pip install*, so also when running *pip install* it is needed to specify the version number 2.6.0.

4.2.2 Application quantization and Dataflow

Regarding the application code, major changes have to be done on the data processing, both on the input and the output, because now the model will only accept 8-bit integer values already quantized, and it will output 8-bit integer output scores. This means that the parameters and thresholds used to compare the model output have to be converted as well, and exactly the same quantization parameters (slope and offset) used to quantize the model operators have to be used to quantize the input data. Dequantization is not necessary here, since it is not possible to go back to floating point arithmetic and also the integer output score can be easily interpreted as it is.

As a result, a new application code flow scheme is needed, adding the quantization steps, as it can be seen in Figure 4.2.

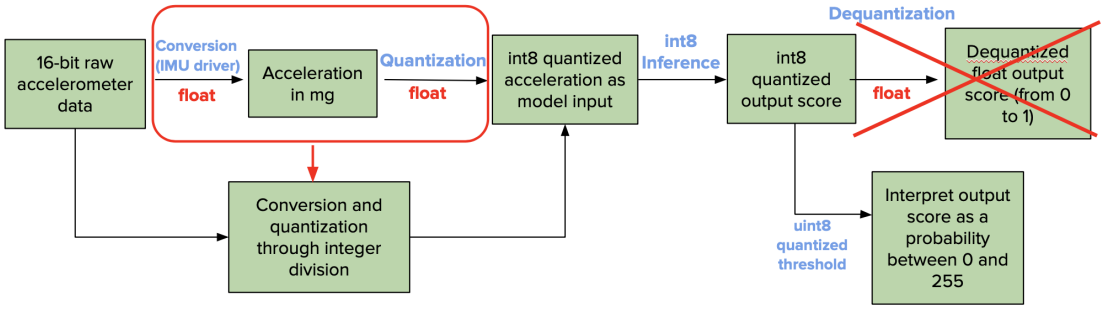


Figure 4.2: *Magic wand*, quantized application code flow scheme.

The quantization parameters are given as an output by the TF Lite converter (and can be also seen in Netron app, or accesses through the C code after importing the model). All the C++ files listed in the previous Section needs to be modified, because all the variable types needs to be *int8_t* and the data processing (mostly the input quantization and the output average) must be done by using exclusively integer operations. In particular, to transform any floating point division into integer, it is needed to multiply both the dividend and the divisor by the appropriate power of ten that makes the operands become integer numbers, and then perform the usual integer division.

Regarding the output processing, the quantized model will return as output four 8-bit signed integer numbers, corresponding to the probabilities of having found each gesture in the input data. In the floating point version these outputs were not modified, but averaged and compared to a single threshold (corresponding to 0.8), while now to make integer average and comparison easier, it has been chosen to convert the output to unsigned 8-bit values by just adding 128, so that a 0 output corresponds to 0 probability, while an output of 255 corresponds to a probability of 1. Finally, the integer average is made just like the normal average, accepting the fact that in C or C++ the integer division will truncate the result, so to partially compensate for this truncation, an offset of N/2 is added to the total sum before dividing by N (N is the number of output scores to be averaged).

A consequence of what above said is that in general the accuracy will decrease both for the input and the output processing, which, together with the noise, is why it has been said that the accuracy of the quantized model given by the TF Lite converter is actually just an ideal accuracy.

The data structure of the algorithm changes a little, and it is summarized in the following Figure 4.3.

The dataflow is described in the table shown in Appendix Figure A.1 (Appendix A), where the application execution is divided in macro-steps and for each step the data involved is described in detail, from the input collected by the accelerometer to the output that triggers the LED lighting up.

A different description of the dataflow can be seen in Figure 4.4, where the same color-code and style of Figure 4.3 is used.

Some considerations can be done now that the dataflow has been clearly explained. First of all, the size of the circular buffer *save_data* could be brought down to 128x3 without changing in any way the dataflow, since only the most recent 128x3 elements are extracted from this buffer at every inference.

Another interesting consideration could be that the number of consecutive outputs to be averaged can be changed to obtain better results terms of both accuracy and application responsiveness, in order to have an output almost immediately after the gesture have been performed. This is particularly useful when the inference is slow and waiting for many consecutive inferences could become unpleasant. As a final remark, the threshold value should be changed with a trial & error technique, in order to have the best result with the available HW. It is also possible to differentiate the threshold value based on the gesture, and this has been done in this case, having a high threshold for the slope gesture (easily recognized) and a lower one for the ring gesture (hard to recognize).

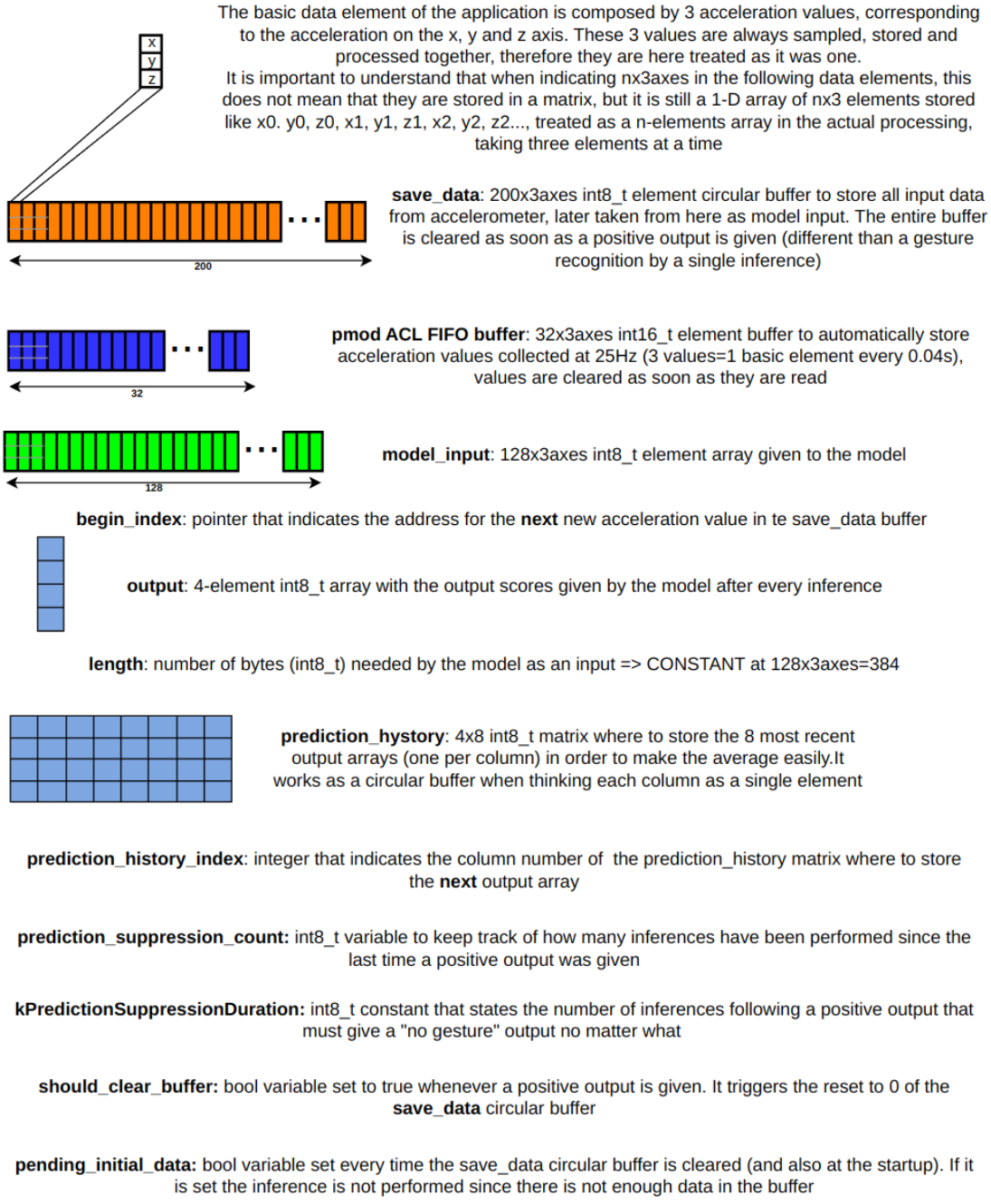


Figure 4.3: *Magic wand*, Data structure.

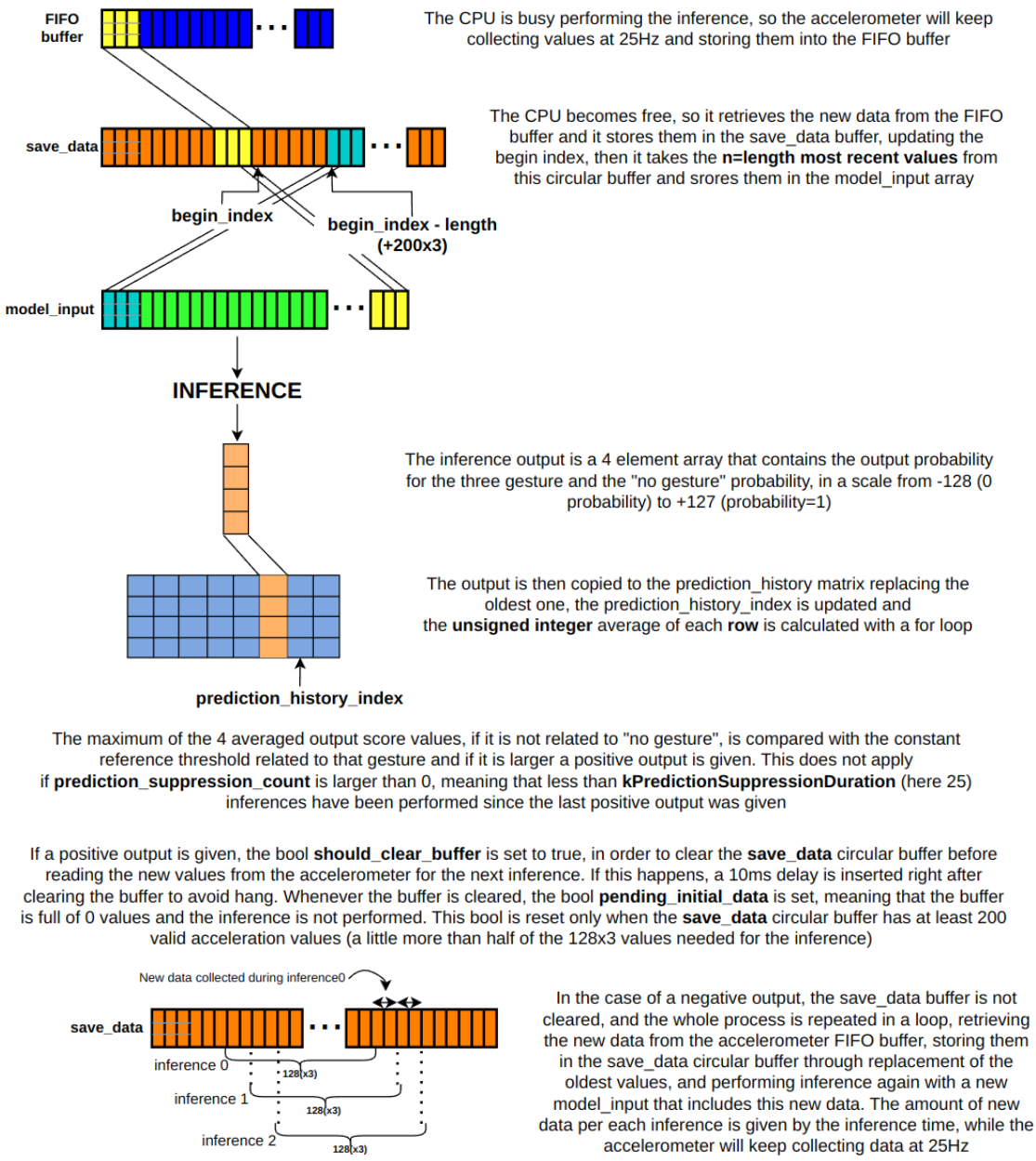


Figure 4.4: *Magic wand*, Dataflow scheme.

4.3 SparkFun implementation with ADXL345

This part of the project is not really required for the porting to RISC-V, but it is very useful as it allows to get familiar with a different accelerometer, so that

when actually using the RISC-V core, the part related to the accelerometer will already be tested. In this case no changes are made on the model, but only the application code that regards the accelerometer is changed, and in particular the file *accelerometer_handler.cc* is replaced by *pmod_acc_handler.cc*, which is very similar, but without using the BSP from the SparkFun. The same happens for the corresponding header files.

The SparkFun Edge board has an I2C port exposed with the QWIIC connector to allow expanding the functionalities by connecting multiple I2C sensors and peripherals. The ADXL345 can be therefore connected to this port and communicate with the Ambiq microcontroller through I2C, just like the embedded accelerometer (Figure 4.5 shows the connection).

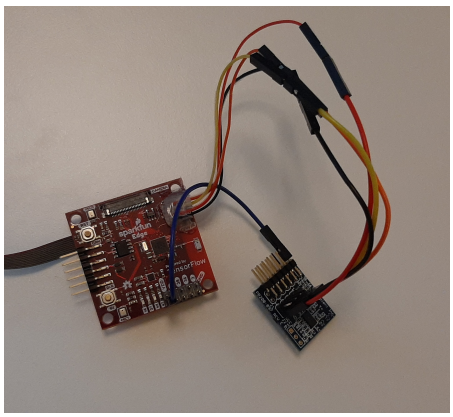


Figure 4.5: SparkFun Edge board - custom connection ADXL345 through QWIIC port.

Since the protocol is the same, the basic functions for reading and writing registers on the sensor can be used as they are from the BSP, while what changes is basically the address and the structure of the internal registers. As a consequence, new small support functions are created to perform the basic initialization operations (such as setting the output data rate, reading the ID register, start the measuring mode, etc.), as well as new address constants and register struct types are declared in the header file. It is also needed to configure, initialize and enable the output pins of the QWIIC connector with the correct settings, which can be found in one of many examples available on the web.

The function related to reading the acceleration values can be left almost the same as in the previous step, with a small change on the data processing due to the different alignment of the output data.

A final remark about this step is that it is expected a slightly lower accuracy when using this new accelerometer, and this is for two main reasons: first of all in high resolution mode the ADXL345 has less data bits than the LIS2HD12 (10 bits

vs 12 bits), but also it is important to consider that the data used to train and test the model has been collected using the LIS2HD12 accelerometer, therefore a slight change in how the data is collected could mean a slight bias on the model output.

4.4 RISC-V implementation

Porting the *magic wand* application to the RISC-V core already means having a first fundamental result for this project: as said at the beginning, the goal is experimenting and understanding the capabilities of the microcontroller, therefore having a NN application running on such a target device already means understanding that it is capable of handling a quite complex algorithm without any issue.

In order to achieve that, some intermediate steps are needed to get familiar with the core, understand how to program it, how the few needed IPs can be properly used, and how to move from an ARM implementation to a RISC-V one. As a consequence, two preliminary tests are run: one to access and use the ADXL345 acccelerometer, while the other one to exploit the internal timer to perform time measurements or make delay functionalities. After that, some work on the application makefile is needed, to have the proper setup and options for the RISC-V compiler toolchain.

4.4.1 ADXL345 test

This test is made much easier by the previous step, when this accelerometer has been connected and used with the SparkFun Edge board. In this case, while the general structure of the test program is the same as in the previous step, most of the support functions to access the sensor need to be written again, taking inspiration from the ones already used from the SparkFun BSP.

What completely changes is the communication with the core, since here SPI is the communication protocol of choice, mainly because some good examples are already available about how to set up the SPI IP on the RISC-V core in use. As a consequence, three new functions are written: *SetupSPI*, *ReadRegSPI* and *WriteRegSPI*. The first function allows to configure and properly use the IP according to the design of the RISC-V microcontroller, accessing the right registers to configure the SPI pins and enable the IP with the right clock frequency, while the other two functions implement the SPI protocol for reading and writing from a master to a slave, knowing the address of the register and, if needed, the value to be written or the number of bytes to be read.

Once these three functions are written, a simple test can be run on the RISC-V FPGA model, to verify that the access and the values read from the accelerometer are correct. After that, the file *pmod_acc_handler.cc* can be modified accordingly,

together with its header file, just adding these new functions and leaving the structure as it is.

4.4.2 RVTimer test

One more test needed is the timer test, used for two main reasons in the application: measuring the inference time and having a simple delay function to introduce a small delay after a positive output is given and the big circular buffer is cleared. Some preliminary information can be useful: only one timer is available and it has a 64-bit counter, it works at 12 MHz (CPU frequency), with the possibility to set a clock prescaler. It can work in output-compare mode and interrupts can be used if needed. In this case interrupts are not used, since to measure the inference time it is just necessary to retrieve the counter value in two different moments, while when a delay is needed, in the specific case of this application the CPU has nothing else to do and therefore it can just wait (polling implementation).

The procedure is similar to the accelerometer test, but in this case some support functions are already provided with the IP. These functions allow to access the counter value, enable the timer and change the value in the output-compare register to implement the delay functionality. As a consequence, to measure inference time it is only needed to call these functions, while a separate test can be made for the delay functionality, which means just calling those support functions (to update the output-compare register and enable the counter) and finally polling on the COMP bit that is automatically set when the compare value is reached.

Once again, after the test is completed, the new functionalities can be added to the application code, by creating a delay function in *pmod_acc_handler.cc* and adding few lines right before and right after the inference to enable the timer in free running mode and later fetch the counter value.

4.4.3 RISC-V Makefile

The application code at this point is ready to be compiled for a RISC-V target, so now it is needed to compile and link for a 32-bit RISC-V (rv32IMC) architecture by using the gcc toolchain. As mentioned in 3.4, the TF makefile structure requires a .inc file related to the target, so in the folder *micro/tools/make/targets* there is one makefile for any general RISC-V target, called *mcu_riscv_makefile.inc*. This .inc file takes care of downloading the proper compiler toolchain, setup all the compiler and linker flags that are specifically required by the target and update the dependencies with the new eventual files that need to be included (such as all BSP source files and headers).

A few considerations should be made before going into the details on how this file needs to be modified. First of all, as already mentioned in 3.4, the

mcu_riscv_makefile.inc that comes with the TF 2.6.0 stable repository contains a bug, which has been fixed in [19], so the fixed file should be used here. Moreover, the default RISC-V implementation on the TF repository is done for the SiFive targets, which means that what has been done for SiFive can be taken as an example or a as a hint to understand and fix eventual problems with the RISC-V target used in this project. Finally, in the latest years and months there has been an increasing interest on development and support of RISC-V cores, therefore the latest TF versions are likely to be much more complete regarding the support for RISC-V cores. As a result, some more hints to follow for this purpose can be found in the more recent versions of TF, still specifically made for SiFive targets.
Having said that, it can be now understood why the file *mcu_riscv_makefile.inc* from one of the latest TF versions (2.10.0) is essential as a main reference to follow for having a successful porting.

The changes to be made to this makefile are several, listed in the following:

- First of all the source and header files that constistute the BSP for the RISC-V core need to be added to the dependency list, used to know which files need to be compiled for the application to work. In particular, these files allow to implement the right functionalities by using the available IPs, such as SPI (*sspim.c*, *sspim_funcs.h* and *sspim.h*), I2C(*twim.c* and *twim.h*), UART (*uart.c* and *uart.h*), GPIO pins (*gpio.c* and *gpio.h*), timer and counter (*rvtimer.c* and *rvtimer.h*) and print functionalities (*printf-stdarg.c*). Other important files that could be needed are *irq_table.h* (in case interrupts are used), *riscv_csr.h* and *tb.h*. It is essential that the file *chip.h* is included here and also in all the application .c files that use in any way those BSP functions, because it declares and defines and all the necessary variables and functions.

  These files are copied in a new application folder and added in the makefile under the *THIRD_PARTY_CC_SRCS* and the *THIRD_PARTY_CC_ HDRS* lists, so that they will be properly treated and compiled.

- One more very important step regards the startup file. Normally, when compiling for ARM targets, the gcc compiler by default selects and uses an internal startup file that usually works well with those kinds of targets. In this case, the custom RISC-V core needs its own startup file, therefore the flag *-nostartfiles* needs to be added among the *LDFLAGS*, to tell the linker to use the custom startup, which has to be compiled and linked just like any other source file. The available startup files are two: one .c file (*startup.c*) and one assembly file (*startup_as.S*) which need to be both included in the *THIRD_PARTY_CC_SRCS* list. These two files are both needed, since they do not do the same thing (the assembly actually calls and jumps to the .c file), and it is important that they have different names, because if not the

compiler will generate two times a .o file with the same name, overwriting the one that was generated the first.

- As already said several times, an edge application like this always faces limitations in hardware capabilities and storage space. This is why it is normal in cases like this to avoid using and including the C standard library, because it would make the code size too large for the target available code memory (only 128kB here). This is why the flag *-nostdlib* has to be added among the linker flags (*LDFLAGS*), so that the linker will not link against the standard library. Of course some reduced version of the standard library is still needed, which is called Newlib and it is the standard for embedded application with these kinds of restrictions. Newlib has again two possible versions and one of them (called *nano*) is even lighter, made by ARM for extremely limited microcontrollers. While the normal Newlib version is automatically enabled by the *-nostdlib* flag, to specifically use the nano version two more flags need to be added, i.e. *-lc_nano* and *--specs=nano.specs*.

- Explicitly linking against Newlib nano is not enough, because when linking a few errors will pop up about several “undefined symbols", meaning that some functions needed by the library are not defined anywhere. These are called *system calls* and are basic functions that the library needs to handle several things, such as file access, threads handling, etc. Some examples are *fstat*, *sbrk*, *close*, *lseek*, *kill*, *getpid*, *isatty*, *read*, *write*, etc. Even though some of them will not be used, the library still needs a definition, which can even be completely empty. What is mentioned above is explained very well in [21].

  Here the last TF version comes in handy, because the SiFive folder automatically downloaded by the latest RISC-V makefile.inc contains all these function definitions for SiFive targets and also includes them in the makefile. Most of them are just stub functions and all the files can be left as they are, just removing the include to the *platform.h* file. The only one that needs to be taken care of is the *sbrk.c*, which increases the size of the heap (used by malloc, for example). In this file two external variables are used, *_end* and *_heap_end*, which are the beginning and the end of the heap section respectively (with the beginning of the heap corresponding to the end of the stack), declared in the linker script. At this point it is just sufficient to change the names of these variables looking at the linker script of the target in use, or to declare and define them in the right place of the linker script in case they are not. Again, the linker scripts from SiFive can be a good reference.

  The.c files related to the system calls are collected in a folder called “libwrap" and to be properly compiled, they need to be added under the *THIRD_PARTY_CC_SRCS* and *THIRD_PARTY_CC_HDRS*. This is actually not

enough, because they still will not be recognized by the linker, unless some new linker flags are added, like it is done in the latest RISC-V makefile.inc, where for each of those new .c files, two new flags are added, i.e. *-Wl,--wrap=syscall_name* and *-Wl,--wrap=_syscall_name*, where *syscall_name* is the name of the system call, that in the SiFive files corresponds with the name of every .c file.

- As a final remark, it is needed to add all the compiler flags that are required by the specific target in use (some of them will declare and define variables needed by the #ifdef in some .c or .h files), or remove some flags that are not compatible with it. Moreover, a map file can be generated to check the changes made.

4.4.4 Last fixes

In this Section the last fixes to have a working implementation on the RISC-V core are detailed. One fundamental thing when testing and debugging is the possibility to print information on screen. In the TF 2.6.0 the application code does not use the classic printf, but is has a separate error handler (the class *ErrorReporter*) that takes care of printing information through the UART interface. Going through the code it can be seen that this error reporter processes the string to be printed and finally calls a function named *DebugLog* that takes the processed string and prints it. This function has a different implementation for every target, and it can be found in the main TF Lite Micro folder (*tensorflow/lite/micro*, where there is a separate folder for every single target type, each of them containing the *debug_log.cc* file). The RISC-V implementation (in *tensorflow/lite/micro/riscv32_mcu/debug_log.cc*) is empty by default and it needs to be implemented for the target in use in order to have a working print functionality. It is sufficient to include a print function from the target's BSP, and in this case there is a function called *putchar_uart* in the *uart.c* file, that simply takes one character and prints it, therefore it is enough to call this function until the character becomes a string termination character. Also, to enable the print functionality through the UART, some new compiler flags are needed, i.e. *-DDEFAULT_UART_FOR_PUTCHAR=IP_UART0_PTR_UT* and *-DUSE_REAL_UART_FOR_PRINTF=1*, which can be added in the *PLATFORM_FLAGS* in the *mcu_riscv_makefile.inc*.

In the case of this project, a separate bash script (*link.sh*) has been created to take care of the linking part. This is because using the TF makefile process, the compile and link phase do not return any errors, but the produced binary is empty, therefore there is some problem with the creation of the .a library file out of all the compiled object files, which makes the linking step produce a useless binary. In this new bash script, all the options and flags are the same as in the makefile, but the .a file is not used, while instead all the needed object files are listed one

by one. Here it is also possible to add the command to generate the map file and the command to convert the binary into ihex (needed by the programmer and the respective python script).

One last important consideration must be done regarding the code and data memory needed on the microcontroller, since one of the greatest limitations in low-power targets is the available memory. Regarding the data memory, the TF structure uses the so called *tensor arena*, where all the variables used during inference will be allocated. In the default code in the TF repository this *arena* is as large as 60 kBytes, therefore, considering that the quantization steps brings the standard data length from 32-bits to 8-bits, it is possible to safely say that any target with 64 kBytes of data memory can execute the application. Regarding the code memory, here more than 64 kBytes are needed, since the binary that will be flashed on the microcontroller has a size of around 114 kBytes in the case of the 32-bit floating point application, and around 90 kBytes in the quantized version. As a result, at least 128 kBytes of code memory are needed.

4.5 Optimized RISC-V implementation

This is the last step of the thesis project, where the focus is on the possibility to improve the performance obtained in the previous step through both HW and SW changes, also with the help of the current research papers on the topic and considering the eventual optimizations already made in other target types, i.e. the ARM CMSIS library, which is open-source.

4.5.1 Layer choice

The first thing to do in this case is finding out where exactly it is most profitable to optimize, because given the limited amount of time, only one aspect of the application can be studied and eventually optimized, while for the other aspects it will be easier to follow what has been done here to get similar results. Since the implemented NN is a sequence of layers, only one of those layers will be studied, and the most reasonable thing is to try to optimize the layer where application spends the most time during inference.

As it can be seen in Figure 4.6, the *magic wand* application consists of 4 types of layers: convolutional, maxpool, fully connected and softmax. To understand where the most time is spent, the GPIO pins can be used, just changing their value when the function that implements that layer is called.

The application code changes are very few, to be done in each of the .cc files located in *lite/micro/kernels* that represent the TF bridge to the actual operators implementations.

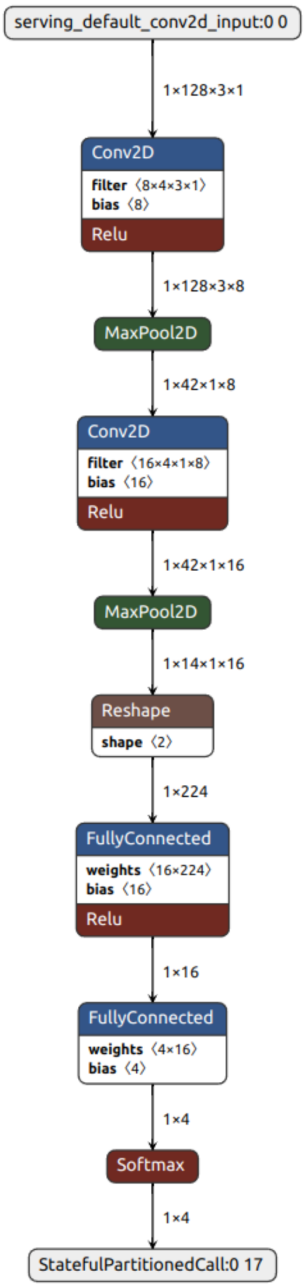


Figure 4.6: Netron app view of the magic wand NN.

The result of this function profiling through GPIO is very clear: the convolutional layer is where almost 90% of the inference time is spent. The measurements have been done both on the SparkFun Edge board (with CMSIS optimizations) and

on the RISC-V FPGA model (non-optimized), and Figures 4.7 and 4.8 show the measurement outcomes obtained using the Digilent Digital Discovery.

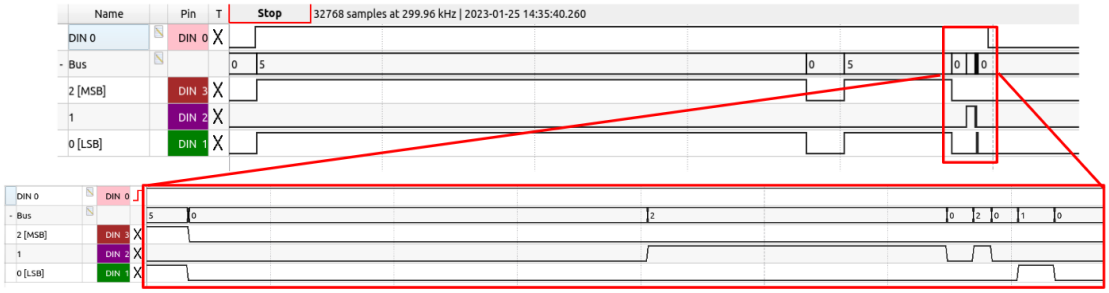


Figure 4.7: GPIO function profiling measurement on SparkFun Edge. DIN0 indicates the inference time. Bus values for each NN layer: Bus=5=> Convolutional, Bus=2=>Fully Connected, Bus=1=>Softmax.

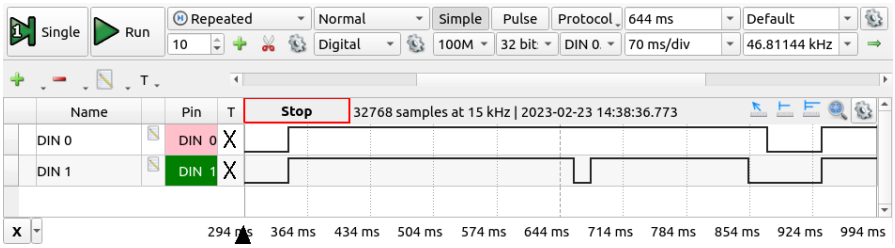


Figure 4.8: GPIO function profiling measurement on the RISC-V FPGA model. DIN0 indicates the inference time, DIN1 is set when executing a Convolutional layer.

It is now clear that the focus needs to be on the convolutional layer, which was also predictable, since this is where the convolution multiplications and additions are performed.

4.5.2 HW/SW optimization choice

At this point the next choice to be done is how to optimize this layer on a HW point of view, and here what said in Section 2.3.2 becomes essential, mostly in a very power-constrained device. The most reasonable option here is to create an in-pipeline AFU, meaning that the ALU will be internally modified by adding in the pipeline an HW block. The idea is not to have the whole layer handled by HW (such as the one described in [17]), but rather to add a small HW block that will perform an optimized MAC operation, which is the time-(and power-)hungry operation continuously used by the convolutional layer. The reference for this idea

is given in [18], where they implemented a parallel MAC for three 8-bit couples of inputs.

In a few words, the goal is to exploit the fact that the quantized application uses only 8-bit integer data in a 32-bit architecture to perform SIMD instructions and therefore obtain performance and time gain through parallel processing. In practice this means to design a new HW block, create and add a new RISC-V instructions that uses that bock and also change the algorithm so that the new block is properly used whenever it is possible.

Before designing, it is important to understand if it is worth it to follow the exact same idea as in [18] or maybe see if a different option can be better. The improvement given by this kind of parallel computation strongly depends on how much it can actually be used, therefore the code that uses this new HW is as important as the HW itself and most likely it also needs to be changed accordingly. In this case, it is very helpful to see what ARM has done in this regards, and how the convolutional layer algorithm is optimized by the CMSIS library, and this can be done by reading the CMSIS algorithm for the convolutional layer (file *arm_convolve_s8.c* in *micro/tools/make/downloads/cmsis/CMSIS/NN/Source*). First of all, the ARM optimization does exactly what has been described so far: it uses SIMD instructions to speed up the convolution operation, and to do that changes the convolution algorithm with respect to the TF internal reference implementation. To allow parallel computation, they implement the Im2col algorithm to perform convolution (explained in Section 2.1.1) and then they perform two 16-bit MACs in parallel, using the 32-bit registers as 2-element 16-bit vector registers and calling a special inline assembly instruction that performs the actual operation (called *SMLAD*). Other inline assembly instructions are introduced here to help with the data manipulation, mainly to go from 8-bit to 16-bit and create these pseudo-vector registers.

The ARM implementation processes two columns of the Im2col algorithm at a time (producing 2 output values) and within each column it performs four MACs at a time, which means calling their SIMD instruction twice (each *SMLAD* calculates two MACs). What just said generates two main considerations: the SW strategy needed to make this algorithm work is quite heavy, not only for the Im2col in itself, but also because this processing in couples for the columns and in groups of 4 for the elements requires extra-care for the cases in which the number of columns and elements within a column are not multiple of 2 or 4 respectively. Moreover, the sign-extension to 16-bit is not really necessary, as well as it is strange to always call this *SMLAD* function twice, instead of having a new instruction that performs four 8-bit MACs instead of two 16-bit ones.

What above said brings to the final decision for the optimizations in this project: the choice is a mix between what done by ARM in the CMSIS library and what done in [18], trying to get the maximum improvement with the available time and

resources. The same Im2col algorithm as ARM CMSIS will be used (also with the columns processed in couples, etc.), while a new HW block will be designed (together with the corresponding new RISC-V instruction) so that it performs **four 8-bit MACs** in parallel, which can be inserted in ARM's algorithm without any change, since their instruction was always called twice in a row. In this way the maximum parallelization is obtained with the available 32-bit architecture and at the same time the CMSIS algorithm can be used as it is without the burden of sign-extending to 16 bits.

4.5.3 HW design

The new HW unit needs to be introduced in the core structure, as part of the ALU, which will use it just like the simple multiply or add unit when the current instruction requires it. The choice is to have this block fully combinational, therefore the result will be given in one clock cycle, with the risk of being in the critical path since the total time needed to get the result equals one 8-bit multiply plus two 32-bit add. In any case, it is still worth it to use this block since the advantages of parallel computation are expected to be very high. Figure 4.9 shows the general scheme of this new HW unit.

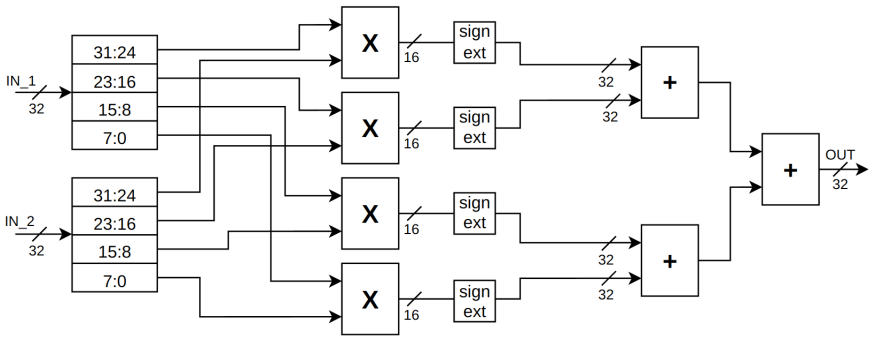


Figure 4.9: New SIMD HW unit with four 8-bit MACs.

The result of each of the first 8-bit multiplications will be a 16-bit signed number, that will be internally sign-extended to 32-bit, so that both the following additions can be performed with a 32-bit parallelism. Overflow is not possible in this case, since in the end the inputs of the adders are always numbers that can be represented in 16 bits. The output of the unit will still be 32-bit, because the accumulation of the result will go on in the algorithm and also in this way the HW unit can be used also for other purposes without the restriction to go back to 8-bit, which in this case will be done via software in the application code.

It would be useful to have a third 32-bit input going into the unit and containing the value accumulated so far, in order to avoid doing one more addition to accumulate with the previous parallel instruction result. Unfortunately, this is not possible with the available HW, since the register file only has 2 read and 1 write ports working at the same time, so instructions with three inputs are not available. This will decrease the gain given by this new SIMD instruction.

As a final remark, the design has been done in verilog and in a behavioral style, while to integrate this unit a new operation selector value needs to be defined in the verilog code for the ALU, and few more lines of code need to be added to indicate that when the operation selector variable holds this new value, then the result of the ALU has to be connected to the output of this new unit. What just said will be better explained in the following section.

4.5.4 RISC-V instruction design

Even if the HW unit design is ready, the block cannot be used until there is an instruction that targets it. In this case, a new RISC-V instruction needs to be designed and included in the core's ISA, together with the mapping of this instruction to the new operation selector value described in the previous Section 4.5.3. Other than that, the compiler also needs to understand the inline assembly instruction that will be in the code and properly map it to this new RISC-V instruction. As a results, both the gcc toolchain and the core code need to be properly changed by adding the support for this new instruction.

First of all, the new instruction has to be designed, choosing its opcode and instruction-type. Having two inputs and one output, it is the classical R-type instruction (see Section 2.2), but before going farther, it is better to see if some similar instructions are already present in some stable or proposed RISC-V extensions, so that it could even be recognized by the compiler toolchain without any modification. As a matter of fact, among the "open" (non stable or ratified) extensions there is the **P** (Packed) extension, that contains all the most important SIMD instructions. Looking at the instructions, there is one that perfectly matches the needs of this project, the *smaqa* instruction. The *smaqa* performs four signed 8-bit MACs and also accepts a third 32-bit input to accumulate with the previous result. Figure 4.10 shows the characteristic of this instruction as they are in the P extension proposal in [22].

The instruction encoding is shown in Figure 4.11.

The idea is now to use exactly this instruction with the same opcode to map the new HW unit in the RISC-V core used here, therefore add only this instruction (and not the whole P extension) to the set of instructions understood by the core. The introduced instruction will have the same name and the same encoding, but it will not do exactly the same thing, since it will only accept two inputs instead of

Mnemonic	Instruction	Operation

SMAQA rd, rs1, rs2	Four signed “8x8” with 32-bit Signed Addition (32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)	a[x] = rs1.W[x]; b[x] = rs2.W[x];
		m0[x] = a[x].B[0] s* b[x].B[0];
		m1[x] = a[x].B[1] s* b[x].B[1];
		m2[x] = a[x].B[2] s* b[x].B[2];
		m3[x] = a[x].B[3] s* b[x].B[3];
		rd.W[x] = rd.W[x] + m3[x] + m2[x]
		+ m1[x] + m0[x];
		
		(RV32: x=0, RV64: x=1..0)

Figure 4.10: Description of the *smaqa* RISC-V P instruction [22].

31 25	24 20	19 15	14 12	11 7	6 0
SMAQA 1100100	Rs2	Rs1	000	Rd	OP-P 1110111

Figure 4.11: Encoding of the *smaqa* RISC-V P instruction [23].

three, as already mentioned in the previous Section 4.5.3. The final accumulate with the previous result must be done just with a common 32-bit add instruction following the *smaqa*.

To do that, it is needed to modify several files on the core's side:

- First of all, the new instruction encoding must be added in the instructions set list (it is a python dictionary in *nanorv32.py*), indicating the instruction type (R), the name and the value of the three fields that map the instruction in the ISA (opcode, func3 and func7).

- The instruction is now in the ISA, but there is nothing that indicates what to do while executing it. This is why also the file *nanorv32_impl.py* has to be modified by adding the new instruction and write what the HW needs to do, such as how the program counter will move, how to connect the ALU's inputs and what operation to perform, as well as if and how the memory and/or the register file are accessed to retrieve the inputs and outputs. In the case of the *smaqa*, the program counter will move to the next instruction (standard increase by 4 bytes), the ALU will perform the *smaqa* operation (now recognized through the new operation selector value), the memory is not

accessed, while the register file is accessed both in read and write mode and connected to the ALU inputs and output.

- If not done already, a new parameter with the name and the value of the new operation selector option needs to be added as a parameter in the parameters verilog file of the core. In the same file, a new parameter with the mask of the new instruction encoding has to be created, specifying the opcode, func3 and func7 fields while leaving the other bits to *don't care* value.

- A tool can be used to automatically update and modify the core verilog file with what written in *nanorv32_impl.py*, but if the automatic tool is not used, it is needed to manually add the verilog code that tells what to do and how to connect the HW blocks when executing this new instruction.

The last step to have a working implementation is to modify the gcc compiler so that it recognizes and correctly maps the new instruction. This procedure has already been mentioned in the background chapter (Section 2.3.2). To do that, the latest version of the gcc compiler has been fully cloned from the GitHub repository and the guide found in [24] has been followed. The guide explains everything very clearly, but the only thing to keep in mind is that it has been made for a 64-bit ISA, so at the moment of linking it will not work. This is why in the configuration step along the process, the line *./configure --prefix=/opt/riscv_custom* has to be replaced by *./configure --prefix=/opt/riscv_custom --with-arch=rv32gc --with-abi=ilp32d --enable-multilib*. In this way the 32-bit version will be used.

As explained in [24], the files *riscv_opc.c* and *riscv_opc.h* in *riscv-gnu-toolchain/binutils/opcodes* and *riscv-gnu-toolchain/binutils/include/opcode* need to be updated by adding the mask and match macro for the new instruction opcode and func3/func7 fields, to allow the compiler toolchain to recognize the instruction without giving any errors.

The final step for a working implementation (after having properly modified the algorithm) is to update the *mcu_riscv_makefile.inc* in order to use this new version of the toolchain, changing the *TARGET_TOOLCHAIN_PREFIX* variable to *riscv32-unknown-elf-* and also adding the path to the bin folder of the new toolchain in the environment variable *PATH*.

4.6 Tests and simulations

In this Section the tests and simulations done during the implementation process will be briefly detailed, to show how the changes made to the SW or to the HW have been verified.

4.6.1 Basic implementation on SparkFun Edge

In the first implementation step, the application has been tested as it is, by running *make -f tensorflow/lite/micro/tools/make/Makefile TARGET=sparkfun_edge magic_wand_bin* from the main tensorflow folder. The only test that was made was just switching from the TF internal operator implementation to the ARM optimized one, through the CMSIS NN library targeted for the Cortex M4 core. This is a very important step, since it is here that the structure of ARM's optimizations has been studied and understood. One fundamental consideration here is that when going from the TF internal implementation to the CMSIS implementation, it is needed to perform a make clean (*make -f tensorflow/lite/micro/tools/make/Makefile TARGET=sparkfun_edge clean*, from the main tensorflow folder).

4.6.2 Model quantization on SparkFun Edge

During the quantization step, the main problem was represented by the TF versioning issue, that has been already described in Section 3.4. Apart from that, the quantization has been done in Python with the TF 2.6.0 library, by modifying the script *train.py* in *micro/examples/magic_wand/train*. The test done in this case was also useful to measure the accuracy of the quantized model, as described in Section 3.5.2. A new python script (*eval_tflite.py*) has been written and data from the test dataset were given as input to the model and the result was checked with the gesture label to calculate the final ideal accuracy, considering the same thresholds used in the application C code to know if the output score is high enough to identify a gesture. To test the application on the SparkFun Edge, after changing the code, the thresholds have been adapted to the new output characteristics (differentiating them between the three possible gestures), and again the internal TF implementation has been compared with the ARM CMSIS one, now with a closer look at the benefits gained in inference time.

4.6.3 SparkFun implementation with ADXL345

Changing the accelerometer does not really modify the application structure, therefore in this case the test has been mostly focused on making the ADXL345 accelerometer work properly, printing on screen the register values and the acceleration measured. Finally, the *pmod_acc_handler.cc* file has been created to integrate the new accelerometer in the application and verify that nothing really changes.

Figure 4.12 shows the test setup for the SparkFun Edge board, with the SEGGER programmer on the left and the ADXL345 accelerometer on the right.

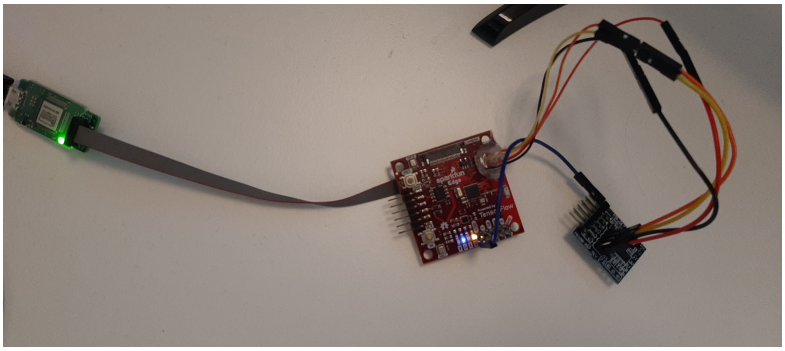


Figure 4.12: Measurement setup - SparkFun Edge board with SEGGER J-Link Edu-mini and Digilent Pmod ACL.

4.6.4 RISC-V implementaton

Testing the application on the RISC-V core was done step-by-step, trying to fix small bugs at each iteration. As said in Section 4.4, first of all the accelerometer and the timer were tested on their own and out of the TF structure. After having a meaningful binary, the application was tested by running *make -f tensorflow/lite/micro/tools/make/Makefile TARGET=mcu_riscv TARGET_ARCH=riscv32_mcu magic_wand_bin* from the main tensorflow folder, but it crashed every time a print function was called. After modifying the *debug_log.cc* file as described in Section 4.4.4, the application was tested by injecting fake input and feeding them to the model, in order to isolate the inference process and neglect every eventual bug coming from the input retrieving and processing. The fake input was taken from a real inference that gave a positive result for the *wing* gesture on the SparkFun Edge with the quantized model. In this way the data is already processed as the model wants it, so it can be understood if the model inference does its job or any operator gets stuck or gives the wrong result.

After that, the application was tested with live acceleration data, to confirm that the input processing works properly, as well as to add inference time measurements. In this case the inference was extremely slow, therefore to get some results the output score average has been discarded and the thresholds values were properly tuned. Moreover, the output suppression count after a positive output was drastically reduced (to 1 or 2 inferences) to avoid having too long waiting time.

Figure 4.13 shows the test setup for the RISC-V core (actually its FPGA model), with the Digilent Arty S7 FPGA board on the left, the Adafruit FT232H programmer at the bottom, the ADXL345 accelerometer in the center on top and the Digilent Digital Discovery on the right.

In Figure 4.13 it can be also seen that the Digital Discovery is connected to the pins of the accelerometers, and this was done as a preliminary test for the SPI

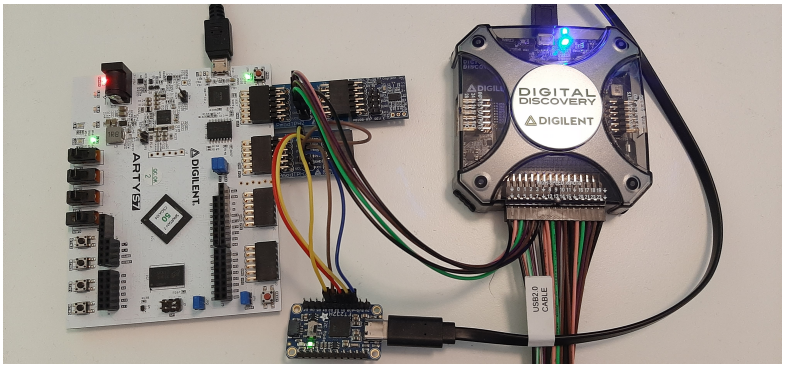


Figure 4.13: Measurement setup - RISC-V FPGA model with Adafruit FT232H, Digilent Pmod ACL, and Digilent Digital Discovery.

communication. The waveforms corresponding to this measurement are shown in Figure 4.14, where it can be seen the SPI transaction for reading the acceleration values from the internal FIFO of the accelerometer. In particular, the first byte sent on the MOSI line is 0xF2 (binary 11110010), with the MSB set to 1 for “read" transaction, the second MSB set to 1 for “multiple byte read", and then the address to be read, which is 0x32 (binary 110010), corresponding to the *datax0* register of the accelerometer. After that, a dummy byte (0x22) is sent for 6 times just to give the clock to the slave (the accelerometer) and read 6 consecutive bytes on the MISO line, corresponding to the three 16-bit values of the x, y and z accelerations (in this order). As it can be seen, the board was in a static horizontal position, because the acceleration values are 0 along the x and y axes, while 0xF2 (decimal 242) along the z axis. From table 3.1 it can be read that at ±2g full scale (like in this case) the sensitivity of the ADXL345 is 4mg/digit, so to convert the read value in mg it is only needed to multiply by 4, obtaining an acceleration of 968mg, which is around 1g (gravity acceleration on the z axis).

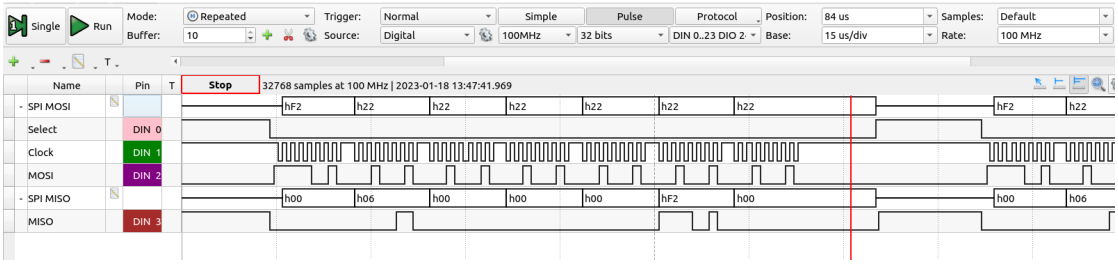


Figure 4.14: SPI read transaction monitored with Digital Discovery to get the acceleration values from the ADXL345.

4.6.5 Optimized RISC-V implementation

In this last step, separate preliminary tests need to be done on the HW and on the SW, since there are important modifications on both sides.

On the HW side, the typical digital design approach is followed: first of all the new HW unit is designed and tested on its own by writing a verilog testbench and running a simulation. Figure 4.15 shows the RTL schematic view obtained from the verilog code using Vivado. Here the structure of the four MACs can be seen, with first four 8-bit parallel multiplier and then three 32-bit adders to accumulate the results. The sign extension block between the multipliers and the adders is not shown here, but its presence can be seen looking at the number of bits in the multipliers outputs and in the adders inputs.

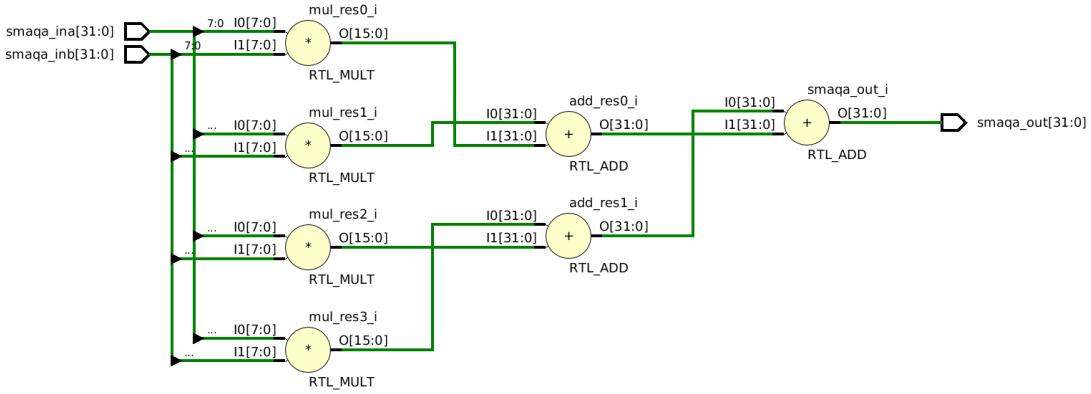


Figure 4.15: RTL schematic view of the HW block in Figure 4.9.

After that, the unit is included in the core's ALU and a small and simple C firmware is written to see if the unit is actually used, as well as to test if both the compiler toolchain and the ALU recognize the new RISC-V instruction both in simulation and on a real FPGA test. In this case a simple 4x4 row by column MAC was performed, to see if the parallel computation gave the right result.

Figure 4.16 shows a snippet of the objdmp performed on the .elf produced by the compiler/linker for this simple C program, and in particular the initial part of the main function, to show how the compiler recognized the new*smaqa* instruction and treated it as any other instruction, giving it the right name recognized by the core.

Figure 4.17 shows the waveforms of the most important signals in the simulation: the two 32-bit inputs (*smaqa_ina* and *smaqa_inb*), the 8-bit inputs to the four multipliers (*a0* to *a3* and *b0* to *b3*), the 32-bit result of the four MACs (*smaqa_out*)

```
~/work/gitlab/icdesign/ips/aldebaran/sim/ctests/simd_mac_test ❯ nn-accelerator ● ?
dalo@onio-ws02  17:51:08  /opt/riscv_custom/bin/riscv64-unknown-elf-objdump -D simd_mac_test.elf
| grep -n -A 80 "<main>:"
125:00000354 <main>:
126- 354:       fe010113                addi    sp,sp,-32 # 4000ffe0 <__HeapLimit+0xfd60>
127- 358:       030205b7                lui     a1,0x3020
128- 35c:       060507b7                lui     a5,0x6050
129- 360:       00112e23                sw      ra,28(sp)
130- 364:       10058593                addi    a1,a1,256 # 3020100 <SYSCALL_ADDR+0x1010100>
131- 368:       40378793                addi    a5,a5,1027 # 6050403 <SYSCALL_ADDR+0x4040403>
132- 36c:       c8f585f7                smaqa   a1,a1,a5
133- 370:       00001537                lui     a0,0x1
134- 374:       be450513                addi    a0,a0,-1052 # be4 <__modsi3+0x30>
135- 378:       00000097                auipc   ra,0x0
136- 37c:       62c080e7                jalr    1580(ra) # 9a4 <printf>
137- 380:       01c12083                lw      ra,28(sp)
138- 384:       00000513                li      a0,0
139- 388:       02010113                addi    sp,sp,32
140- 38c:       00008067                ret
```

Figure 4.16: Snippet of the objdmp of the .elf file of the test program with the new *smaqa* instruction at address 36C.

and finally the value of the ALU operation selector, that corresponds to the value assigned to the new *smaqa* instruction (50 in this case).

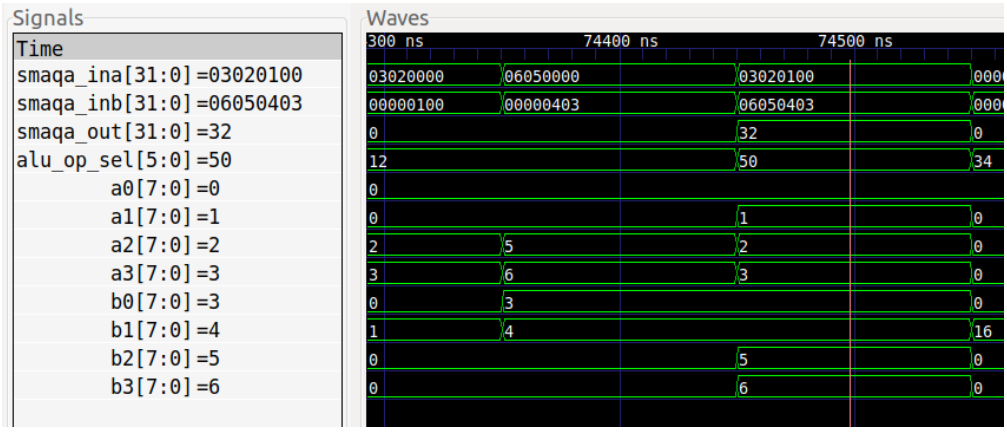


Figure 4.17: Waveforms from the simulation of the new *smaqa* HW block.

Once the HW has been simulated and tested, the SW test can be run, and in this case the algorithm in itself was entirely taken from the CMSIS Im2col implementation, but it was using ARM custom types for the variables, as well as several custom RISC instructions concerning 16-bit parallel computation. The first step is to create a C program that emulates this convolution algorithm in an environment that is independent from TF, so that the HW unit and the new algorithm can be tested together in complex convolution operation, without the heavy background structure and system flow coming from TF. This has been done by discarding all the unused variables and types and leaving only the input data, the convolution filters and few essential convolution parameters (stride, padding, offset and shift). To automatically check the result, the simple convolution algorithm

coming from TF internal reference implementation was modified in the same way and included in this test program. In this way full simulation and test were performed to verify the correct functioning of both the algorithm and the HW. Figure 4.18 shows the input values, the filter values and the output from the two algorithms.

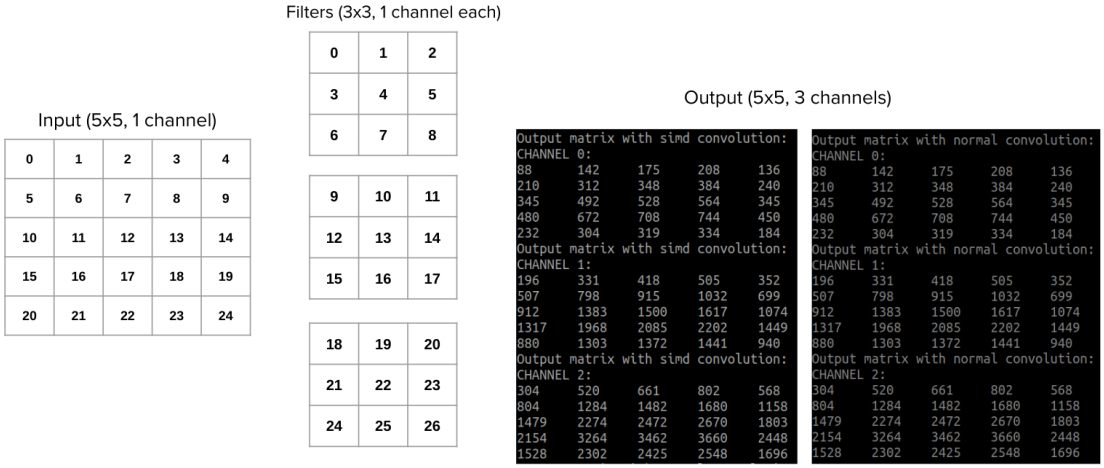


Figure 4.18: Convolution test out of the TF environment - input, filters and output values.

Finally the test on the TF application was performed. The ARM custom types have been replaced by standard types and some basic TF functions and types were used to match code structure and function call present in the TF internal operator implementation. In a few words, the function call and some support functions such as quantization, saturation and cast to 8-bit need to be in TF-style, while the algorithm is in CMSIS-style.

Chapter 5

Results

In this chapter, the results obtained along the thesis project are shown in detail. Since the research questions stated in Section 1.2 and the related purpose and goals (Sections 1.3 and 1.4) are actually not expressed in a quantitative way, the tests described in the previous sections already demonstrate qualitatively that the constrained RISC-V microcontroller is capable of running a simple, but complete NN and that it can even be fast enough for practical applications if the rights HW and SW changes are applied. On the other hand, it is also important to state quantitatively how well the application would perform in a practical case where the studied application is needed, both in terms of accuracy and power consumption, and this is why some measurements have been performed.

As already said in Section 3.5, the measurements have been performed on four areas: the number of instructions executed during inference (strongly related to the power consumption), the inference time, the model accuracy and the size of the binary.

Table 5.1 shows the results obtained by measuring the inference time in all the possible cases with the internal timers embedded in the boards. The measurements have been verified by using the Digilent Digital Discovery connected to some GPIO pins, as well as by using the *mcycle* CSR on the RISC-V core (as described in 3.5.2 and shown in table 5.3) and the DWT unit on the ARM core (table 5.4). Repeated measurements on consecutive inferences always show the same result, meaning that the variation in the inference time is negligible with respect to the ms scale that is used here to compare the results.

The ideal **model accuracy** calculated by SW during the training process and related to the TF 32-bit floating point model (still not converted to .tflite) is equal to **0.89**.

After converting to the flatbuffer format and to the .tflite extension, the 32-bit floating point model has an accuracy of **0.84**, calculated by the TF Lite converter.

Table 5.1: Inference times measurements in all the studied cases, with ratios between some relevant ones.

	SparkFun Edge		RISC-V core	**Clock Ratio**
Clock Frequency	96 MHz		12 MHz	8:1
Model/computation type	32-bit Float	Quantized 8-bit Int	Quantized 8-bit Int	**8-bit Ratios**
TF internal implementation	31 ms	66 ms	520 ms	1:7.9
Optimized (CMSIS or custom RISC-V)	31 ms	9 ms	140 ms	1:15.6
Ratios	1:1	7.3:1	3.7:1	

Finally, the quantization step to a fully 8-bit integer model brings the ideal accuracy down to **0.79**, calculated by the custom python script as described in Section 3.5.2. Table 5.2 shows these results one close to the other.

Table 5.2: Ideal accuracy values in the three optimization steps.

	TF 32-bit model	.tfile 32-bit model	.tflite 8-bit model
Ideal accuracy	0.89	0.84	0.79

The **binary size** is related to the .tflite models before and after quantization and it is calculated by the TF Lite converter. The model before quantization (32-bit floating point) has a binary size of **19616 Bytes**, while the quantized model (8-bit integer) has a binary size of **8656 Bytes**. One more thing that needs to be considered is the size of the whole application binary code, which is the size of the binary file that will be actually flashed on the target device, strongly related to the minimum code memory required on the target to have the application running. In the case of 32-bit floating point model, the final binary size is around **114 kBytes**, while for the quantized application it is around **90 kBytes**.

Table 5.3 shows the results of the **number of instructions** executed, the number of cycles and the CPI during every single convolutional layer and for the whole inference, related only to the RISC-V implementations (optimized and non-optimized) and calculated using the CSRs (see Section 3.5.2). The values concerning the number of cycles have been checked and confirmed by measuring them with the Digital Discovery through the GPIO function profiling described in

4.5.1. Several measurements on consecutive inferences show a negligible variance, always around 0, since the values are less than 100 cycles or instructions far from to the averages listed on table 5.3.

Table 5.3: Number of instructions and cycles measurements for every single convolutional layer and for the whole inference in the RISC-V core, with ratios between non-optimized and optimized implementation.

		Optimized	Non-optimized	**Ratios (non-opt/opt)**
1st conv layer	n. of instructions	609300	3274100	5.37
	n. of cycles	847000 (70 ms)	3765000 (314 ms)	4.45
	CPI	1.39	1.15	0.83
2nd conv layer	n. of instructions	194470	1822980	9.37
	n. of cycles	246770 (20 ms)	2077400 (173 ms)	8.42
	CPI	1.27	1.14	0.9
whole inference	n. of instructions	1116934	5408900	4.84
	n. of cycles	1691400 (140 ms)	6262200 (520 ms)	3.7
	CPI	1.52	1.16	0.81

The same can be done with the ARM core in the SparkFun Edge board, by using the DWT unit as explained in 3.5.2. The obtained results are shown in table 5.4.

To have a better understanding on the optimization advantages and comparison between ARM core and RISC-V core, Figures 5.1 and 5.2 give a visual representation of the data in tables 5.3 and 5.4 through histograms.

Table 5.4: Number of instructions and cycles measurements for every single convolutional layer and for the whole inference in the SparkFun Edge, with ratios between non-optimized and optimized implementation.

		Optimized	Non-optimized	**Ratios (non-opt/opt)**
1st conv layer	n. of instructions	693409	4265641	6.15
	n. of cycles	693408 (7.2 ms)	4265640 (44 ms)	6.15
	CPI	1	1	1
2nd conv layer	n. of instructions	135048	1927671	14.27
	n. of cycles	135047 (1.4 ms)	1927670 (20 ms)	14.27
	CPI	1	1	1
whole inference	n. of instructions	911595	6423100	7.05
	n. of cycles	911579 (9 ms)	6423100 (66 ms)	7.05
	CPI	1	1	1

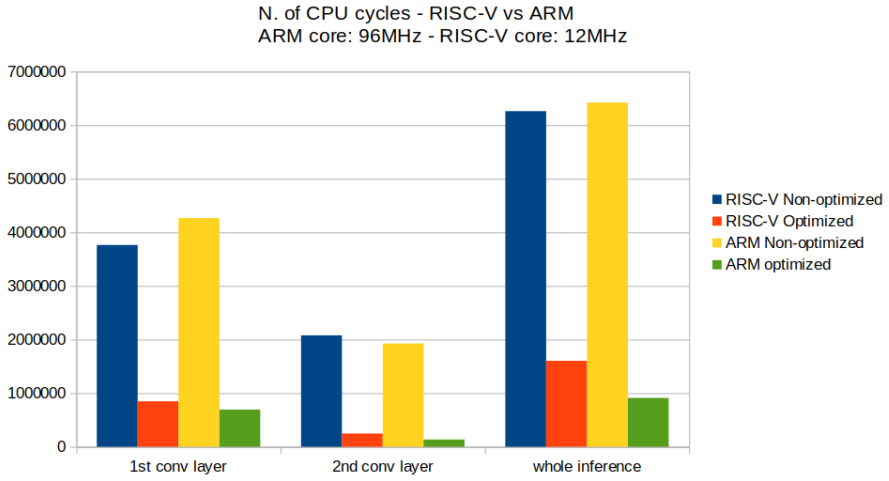


Figure 5.1: Histogram showing the number of CPU cycles as in tables 5.3 and 5.4.

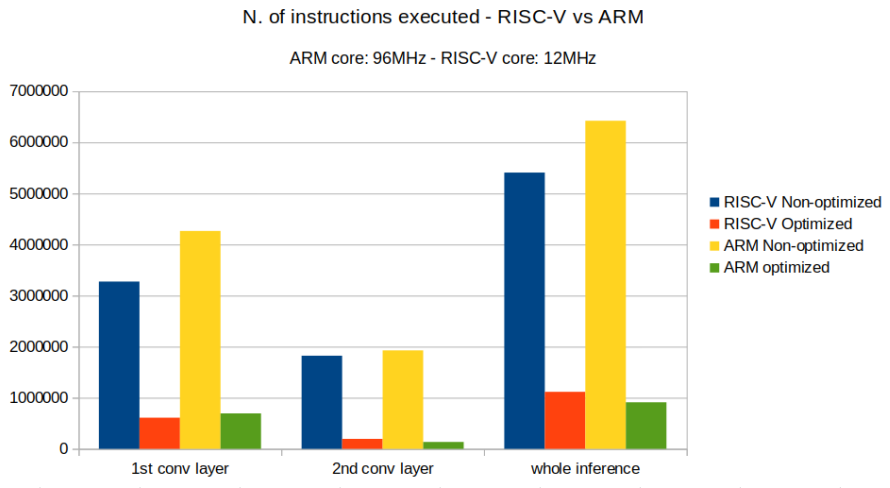


Figure 5.2: Histogram showing the number of instructions executed as in tables 5.3 and 5.4.

Chapter 6

Discussion

In this chapter the results previously shown will be discussed, explaining why those numbers have been obtained and what is their meaning in the point of view of this thesis project.

The **inference time** measurements seen in table 5.1 are a very important to understand quantitatively how good the optimization is. As said in Section 3.5.2, both the timers used to measure the inference time on the SparkFun Edge and on the RISC-V core run at the same frequency, 12MHz, therefore they have the same accuracy and the comparison can be made without considering problems deriving from different accuracies.

It is important to take into account the several ratios listed in the table: in particular the clock frequency ratio needs to be the starting point, because if that difference is not considered, then the RISC-V core performance would seem too poor without a real explanation. As a matter of fact, the SparkFun Edge board has a microcontroller that runs at a frequency that is 8 times higher than the one of the RISC-V core, and it is expected to find the same ratio in inference time between the two boards when they run the exact same algorithm with the same TF internal reference operator implementations. This is exactly what happens, since the inference time ratio for the quantized 8-bit integer model with TF internal implementation is 7.9.

On the other hand, it is interesting to see the improvements brought to the inference time by the different optimizations studied in this project, i.e. the ARM CMSIS library and the custom optimization on the RISC-V core. ARM was capable of obtaining an inference time 7.3 times faster with respect to the TF implementation, while the optimized version of the RISC-V core was “only" 3.7 times faster. The reason behind that is that the CMSIS library optimizes every single layer of the neural network, while in this project only the convolutional layer was optimized (see Section 4.5.1). Moreover, in the ARM implementation

the memory access is well optimized by using their custom *memcpy* function to access the memory and put the 8-bit values into a 32-bit variable, while in the current RISC-V implementation *memcpy* was not used, but a C *union struct* was deployed, accessing and copying the 8-bit values one by one. Finally, the new Im2col algorithm on one hand allows SIMD operations, but on the other requires a heavy data movement, with many duplicated values and with data transferred to temporary intermediate buffers, therefore an improvement of 3.7 times is quite a good result, considering that the memory access is much less optimized if compared to the CMSIS implementation.

Regarding the **model accuracy**, the obtained values are in line with the expectations, because it can be seen that at every conversion/size optimization step around 5% of accuracy is lost. Considering that the initial accuracy was very high (almost 90%), a 10% loss is acceptable, also because the advantages in terms of model size and computational load are essential for the practical application in this specific case. Once again, it is important to state that this is only the ideal accuracy (see Section 3.6), therefore when the application algorithm is introduced (with noise, average and data processing/quantization) it is expected to have a further drop in the accuracy. In the case of this project the goal is not exactly to have a fully functional application ready to be used in a practical case, therefore the loss in accuracy is considered secondary, as long as the application can be properly tested.

The **binary size** of the model has been reduced to less than a half by the quantization process, which is not as much as expected, since ideally going from 32-bit to 8-bit variables the size should become 1/4, but of course this is not true, since most of the computation and the intermediate reults in the operators is kept at 32 bits, while the inputs, the outputs and the weights of the convolutional layers are quantized to 8-bits.

Considering the whole application, there is a significant difference of around 34 kBytes between the non-quantized and the quantized version, which makes sense considering that also most of the variables used to collect and store the data are now quantized.

Several important considerations can be done about the results contained in table 5.3. First of all, it is significant that the inference times and their related ratio is coherent with what measured by using the timers (table 5.1). Moreover, here it can be noticed how good the improvement is in the optimized layer only, by looking at the number of instructions executed that is drastically reduced thanks to the new SIMD instruction and also to the new algorithm, that optimizes the data handling of the whole layer. On the other hand, the number of cycles in each

layer is still reduced, but with a lower ratio, resulting in a higher CPI value for the optimized implementation. This is because while the data processing has been optimized, the data access remained practically the same, and the the processing instructions usually take 1 clock cycle to execute, while the data access instructions most of the times require more than 1 clock cycle. As a result, by reducing the number of 1-cycle instruction through the optimizations, the longer instructions start showing more clearly and becoming more relevant on the final CPI value, which is therefore increased by the optimizations applied in this case.

One more interesting thing that can be seen in this table is the difference in the optimization gain on the two convolutional layers of the application, which demonstrates how the effect of new SW and HW design actually strongly depends on the structure of the convolution itself in each practical case. Here there is a reduction in the number of instructions of 9.37 times in the second convolutional layer, while the first one had “only" 5.37 times less instructions. This is mainly due to the fact that in the second convolutional layer the input data is not actually a matrix, but a column vector, therefore the data to be retrieved will always be adjacent to the previous ones (see Figure 2.13). Moreover, the filters in the second layer have exactly the shape that allows the use of the new instruction, 4x1, and also the fact that the number of channels is always a multiple of 4 makes the usage of the new RISC-V instruction particularly convenient, but this is true also for the first convolutional layer.

Comparing table 5.3 with table 5.4, the considerations made earlier in this Section about the inference time are confirmed, since the ratios between RISC-V and ARM are preserved and it can be noticed how the ARM improvement is always a bit better with respect to the RISC-V case, reaching a final improvement of around 7 times in the inference time. What makes this data interesting is the fact that the ARM implementation always manages to have a CPI around 1, meaning that the memory access is very optimized and there are almost no instructions that need more than one cycle to be executed. It is also worth it to mention the fact that bot the number of cycles and of instructions executed by the ARM core is almost always higher than in the RISC-V case, but the difference in the clock frequency is so large that the inference time is much lower in the the SparkFun Edge board.

A final consideration can be done about the mesurements and how the algorithm and the data structure can or cannot influence them, trying to formulate a general guideline on how to tune the application parameters.

The measurements done and the results obtained strongly depend on how the initial default algorithm and its data flow has been changed, so it is worth it to

try to make some general considerations about what correlates the duration of the inference, the data acquisition, the inference speed and the eventual average on several model outputs.

The starting point is the fact that the model has been trained with data being collected at 25 Hz by the accelerometer, therefore it is capable to work properly only with this spacing between input data. As a result, to change this value and have a faster input data stream, the model needs to be trained again with new data, and to do that a whole new dataset must be collected with almost the same conditions and considerations done about the currently used dataset (see Section 3.5.2). However, as already stated several times, the final goal of this project is not to go deep in the NN model and training themselves, therefore this input frequency of 25 Hz will be seen as a constant value that cannot be changed. A similar preliminary consideration can be done about the number of model input data (128x3 axes), since the model “cannot" be changed.

Going to the numbers, looking at the Figure 6.1 it can be seen that one inference will always consider 128 values on each axis, so knowing that each of these values is collected every 0.04 s (25 Hz), it is possible to say that a single inference analyzes a total of 5.12 s of movement.

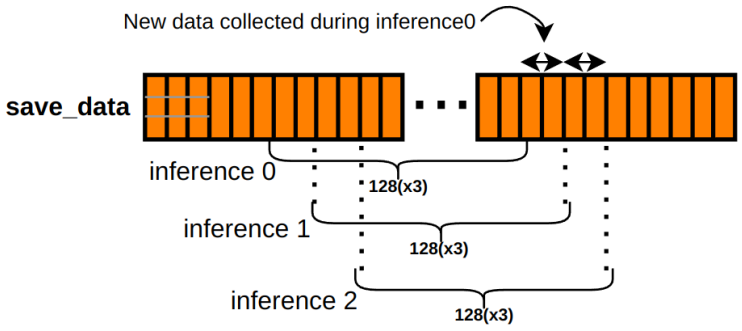


Figure 6.1: General dataflow of consecutive inferences.

Considering that the gestures recognized by the model are about 1s long, it can be seen that one inference covers much more than a single gesture, therefore a gesture must be recognized at any point of that 5.12 s span, as well as multiple gestures could be included in the same inference. What just said is not actually a problem, considering that the inference time is very fast and consecutive inferences operate on almost the same data (see again Figure 6.1), just deleting the oldest values and adding the new ones collected since the last inference. As a result, one single gesture (about 1 s) will gradually move away in the 5.12 s span as the circular input buffer moves and it will be analyzed many times in consecutive inferences. Knowing that, there is no point in having an inference much faster than

0.04 s, because in that case several consecutive inferences will have exactly the same input and therefore the same output: it is just a waste of power and energy. On the other hand, having inferences that last 1 s or 2 s would affect too much the responsiveness of the system and also will not allow to have one gesture in more than 2-5 consecutive inferences. The ideal would be to have one inference every few (for example 1 to 5) new input values, meaning between 0.04 s and 0.2 s, so that even averaging several output scores, the final expected output is given within 1s. In any case, depending on how responsive the application needs to be, also 1 s could be too much, so in that case the number of averaged output can be changed, or some optimizations can be made to have a faster inference and remain with the same number of averaged values.

As an example, the application on the SparkFun Edge with floating point arithmetic has a 32 ms inference, meaning that at every inference there is roughly only 1 new input value per axis in the circular buffer, and this makes every inference having a slightly different output and it allows to average 8 consecutive output scores with almost no delay seen by the user, since 8*32 ms=0.256 s are waited at most to have an answer after a gesture has been performed.

In general, having a fast inference helps because it allows to average many outputs containing the same gesture but in slightly different positions, which reduces a lot the false positive and false negative cases, increasing the final accuracy. What said in this paragraph should help in tuning the average and optimization constants in order to meet the application specific goals.

Chapter 7

Conclusions and Future work

In this chapter some final considerations are done about the work done, going over the goals and objectives listed at the very beginning and trying also to say what could be done better and what are the next steps for a more optimized and final implementation.

7.1 Conclusions

This thesis project tried to combine the research fields of RISC-V microarchitecture and DL, focusing on the low-power edge inference of a NN in order to lead the way to a cutting edge technology that exploits self-sufficient microcontrollers with DL capabilities.

The methodology that guided the project was a simple but effective incremental approach, going from ready made and fully supported examples to tailor-made optimizations on a custom target device, passing by several intermediate steps that helped to gradually get more familiar with the field, the HW and the SW.

In the end, it was demonstrated how a simple but complete NN application can be ported on an extremely constrained RISC-V core and properly work, even though without optimizations it would not be suitable for a practical use, since the inference would be too slow. Moreover, it was seen how both the SW and the RISC-V HW can be modified in order to heavily optimize the NN inference, making it more suitable for a a real practical application and to meet the low-power requirements of a batteryless microcontroller.

The collected data confirmed what above said and clearly showed the improvements given by the optimizations on the constrained RISC-V core, with a 3.7 times reduction on the inference time and a 4.84 reduction on the number of instructions

executed during inference, all with an acceptable drop of 10% in the ideal accuracy. Even though no power measurements was performed, the reduction in the number of CPU cycles and instructions executed definitely implies a strong reduction in the power and energy needed for each inference.

The key that allowed to get the results was focusing all the final efforts only on the most demanding NN layer, therefore the function profiling has been essential to find out where most of the time was spent. On this aspect, it was also of fundamental importance having carefully studied and tested the ARM CMSIS implementation on the SparkFun Edge board, which represented the reference to get inspiration from and to compare the results with.

A final lesson learned can be seen in the method applied during the project and in the true effectiveness of the incremental approach, trying to move step by step from something known and tested to something more experimental, always avoiding the trial&error technique and testing every adjustment before going to the next one.

7.2 Future work

Even though the main goals of the project have been met, there is still a lot to do to make this particular application or any similar one work in the best way possible. As a matter of fact, due to lack of time, not all the aspects have been analyzed and optimized. In particular, as a natural continuation of the work done in this project, three main areas can be studied and further optimized:

- First of all the optimization work done on the convolutional layer can be improved by working on the HW of the core. As an example, the register file of the core can be modified to allow three read accesses at the same time, so that the real *smaqa* instruction from the RISC-V P extension can be used and the four parallel MACs done with just one instruction will include the accumulate operation with the previous result. Even less instructions will be needed in this way.

  Other instructions from the P extension can be analyzed to see if they could be useful in this project, or in any similar application, in order to expand the 8-bit SIMD capabilities of the core.

- Since only one layer has been optimized here, some new HW units or maybe new algorithms can be designed for the other layers, such as MaxPool, Fully-connected and Softmax. In this case, the same procedure and methodology can be easily applied for the convolutional layer, taking inspiration from the CMSIS operator implementations. The advantages are expected to be much

less influential if compared to the convolutional layer, since the time spent on those layers is around 10% of the total inference time.

- Finally, a great improvement can be obtained on the number of CPU cycles by optimizing the memory access and avoid misaligned access to the memory. This issue can be solved by designing some optimized implementation of the *memcpy* and *memset* C functions, just like it has been done in the ARM code. In this way the CPI will probably decrease and the overall inference time and power consumption will benefit from that.
- One more long term goal could be to expand the capabilities of the core by adding SW and HW implementations targeted to quantized DSP, in order to perform NN inference also in those applications that need a heavier input processing, such as the keyword recognition.

7.3 Reflections

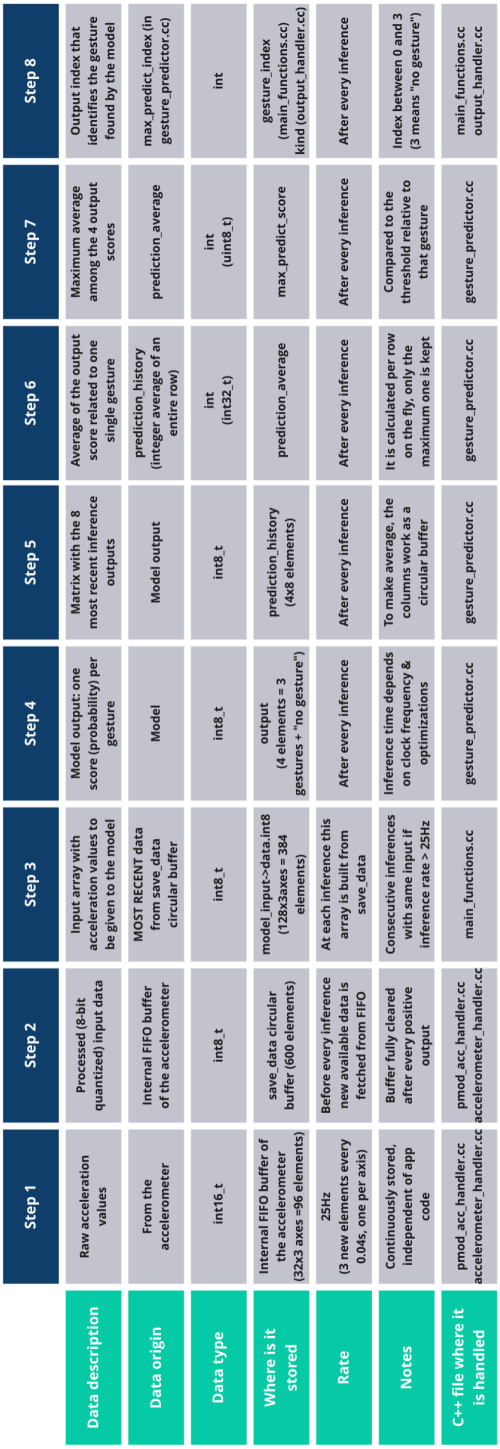
As already stated in chapters 1 and 2, the this project aims at exploiting all the advantages that come from being ultra-low power and from performing NN edge inference. This advantages concern not only merely the technical field, but expands to social, ethical and economical aspects. As a matter of fact, being able to perform NN inference on such a simple and constrained device opens the doors to practical applications on low power microcontrollers that are batteryless and based on energy harvesting, with all the economical and environmental benefits regarding the cost of the microcontroller itself (now very simple and cheap), the waste of resources to produce and replace batteries, the pollution coming from disposable batteries, the maintenance costs needed for checking the battery status and eventually replacing it. On the other hand, considering the fact of performing inference on the edge, several ethical considerations can be made, since here cloud computing is not needed and the large amount of eventually private and personal data needed for the inference will only be stored locally and most importantly temporarily, without sharing it with any other device. In this way privacy is guaranteed.

As a result, this thesis contributes to the UN SDGs numbers 9 (industry, innovation and infrastructure), 7 (affordable and clean energy) and 11 (sustainable cities and communities).

The final hope is that this thesis project will be just the start of a process that will integrate together the most advanced and promising research fields in HW and SW (RISC-V and ML), generating a truly *embedded system* that will be able to do new, exciting things in a clean, smart and secure way.

Appendix A

Magic Wand dataflow



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
Data description	Raw acceleration values	Processed (8-bit quantized) input data	Input array with acceleration values to be given to the model	Model output: one score (probability) per gesture	Matrix with the 8 most recent inference outputs	Average of the output score related to one single gesture	Maximum average among the 4 output scores	Output index that identifies the gesture found by the model
Data origin	From the accelerometer	Internal FIFO buffer of the accelerometer	MOST RECENT data from save_data circular buffer	Model	Model output	prediction_history (integer average of an entire row)	prediction_average	max_predict_index (in gesture_predictor.cc)
Data type	int16_t	int8_t	int8_t	int8_t	int8_t	int (int32_t)	int (uint8_t)	int
Where is it stored	Internal FIFO buffer of the accelerometer (32x3 axes =96 elements)	save_data circular buffer (600 elements)	model_input->data.int8 (128x3axes = 384 elements)	output (4 elements = 3 gestures + "no gesture")	prediction_history (4x8 elements)	prediction_average	max_predict_score	gesture_index (main_functions.cc) kind (output_handler.cc)
Rate	25Hz (3 new elements every 0.04s, one per axis)	Before every inference new available data is fetched from FIFO	At each inference this array is built from save_data	After every inference	After every inference	After every inference	After every inference	After every inference
Notes	Continuously stored, independent of app code	Buffer fully cleared after every positive output	Consecutive inferences with same input if inference rate > 25Hz	Inference time depends on clock frequency & optimizations	To make average, the columns work as a circular buffer	It is calculated per row on the fly, only the maximum one is kept	Compared to the threshold relative to that gesture	Index between 0 and 3 (3 means "no gesture")
C++ file where it is handled	pmod_acc_handler.cc accelerometer_handler.cc	pmod_acc_handler.cc accelerometer_handler.cc	main_functions.cc	gesture_predictor.cc	gesture_predictor.cc	gesture_predictor.cc	gesture_predictor.cc	main_functions.cc output_handler.cc

Figure A.1: *Magic wand*, Table describing the dataflow.

Bibliography

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740 (cit. on pp. 8, 9, 15).

[2] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly, 2020. ISBN: 9781492052043. URL: https://books.google.no/books?id=sB3mxQEACAAJ (cit. on pp. 10, 11, 27, 28, 54).

[3] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. «DeLTA: GPU Performance Model for Deep Learning Applications with In-Depth Memory System Traffic Analysis». In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 293–303. DOI: 10.1109/ISPASS.2019.00041 (cit. on pp. 12, 14).

[4] Dr. Lachit Dutta and Swapna Bharali. «TinyML Meets IoT: A Comprehensive Survey». In: *Internet of Things* 16 (2021), p. 100461. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2021.100461. URL: https://www.sciencedirect.com/science/article/pii/S2542660521001025 (cit. on pp. 14, 15).

[5] Yi-Ru Chen, Hui-Hsin Liao, Chia-Hsuan Chang, Che-Chia Lin, Chao-Lin Lee, Yuan-Ming Chang, Chun-Chieh Yang, and Jenq-Kuen Lee. «Experiments and optimizations for TVM on RISC-V Architectures with P Extension». In: *2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 2020, pp. 1–4. DOI: 10.1109/VLSI-DAT49148.2020.9196477 (cit. on pp. 19, 33–35).

[6] Peter Goldsborough. *A Tour of TensorFlow*. 2016. DOI: 10.48550/ARXIV.1610.01178. URL: https://arxiv.org/abs/1610.01178 (cit. on p. 19).

[7] *Protocol Buffers, Overview - Google*. URL: https://developers.google.com/protocol-buffers/docs/overview (cit. on p. 20).

[8] *Train a Simple TensorFlow Lite for Microcontrollers Model*. URL: https://colab.research.google.com/github/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/hello_world/train/train_hello_world_model.ipynb (cit. on p. 20).

[9] Robert David et al. «TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems». In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 800–811. URL: https://proceedings.mlsys.org/paper/2021/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf (cit. on pp. 20, 21).

[10] *What Is an Instruction Set Architecture? - ARM*. URL: https://www.arm.com/glossary/isa (cit. on p. 23).

[11] Andrew Waterman and Krste Asanovi´c. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, RISC-V Foundation*. May 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (cit. on p. 24).

[12] *Micro Speech Training - Github*. URL: https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech/train (cit. on p. 25).

[13] *Making the Famous Magic Wand 33x Faster*. URL: https://neuton.ai/news/projects/84-making-famous-magic-wand-33x-faster.html (cit. on p. 29).

[14] Vaibhav Verma, Tommy Tracy II, and Mircea R. Stan. «EXTREM-EDGE—EXtensions To RISC-V for Energy-efficient ML inference at the EDGE of IoT». In: *Sustainable Computing: Informatics and Systems* 35 (2022), p. 100742. ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2022.100742. URL: https://www.sciencedirect.com/science/article/pii/S2210537922000749 (cit. on pp. 29–31, 35, 37).

[15] Marcia Sahaya Louis, Zahra Azad, Leila Delshadtehrani, Suyog Gupta, Pete Warden, Vijay Janapa Reddi, and Ajay Joshi. «Towards deep learning using tensorflow lite on risc-v». In: *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*. Vol. 1. 2019, p. 6 (cit. on pp. 30, 34).

[16] Fen Ge, Ning Wu, Hao Xiao, Yuanyuan Zhang, and Fang Zhou. «Compact Convolutional Neural Network Accelerator for IoT Endpoint SoC». In: *Electronics* 8.5 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8050497. URL: https://www.mdpi.com/2079-9292/8/5/497 (cit. on p. 30).

[17] Zhenhao Li, Wei Hu, and Shuang Chen. «Design and Implementation of CNN Custom Processor Based on RISC-V Architecture». In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, pp. 1945–1950. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00268 (cit. on pp. 31, 32, 69).

[18] Ross Porter, Sam Morgan, and Morteza Biglari-Abhari. «Extending a Soft-Core RISC-V Processor to Accelerate CNN Inference». In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2019, pp. 694–697. DOI: 10.1109/CSCI49370.2019.00130 (cit. on pp. 31, 33, 70).

[19] *TF Lite Micro - RISC-V makefile bug fix*. URL: https://github.com/tensorflow/tflite-micro/pull/321 (cit. on pp. 43, 64).

[20] *ARM developer - How can I count the number of instructions executed by the processor in a given time interval?* URL: https://developer.arm.com/documentation/ka001253/latest (cit. on p. 50).

[21] *From Zero to main(): Bootstrapping libc with Newlib*. URL: https://interrupt.memfault.com/blog/boostrapping-libc-with-newlib (cit. on p. 65).

[22] *RISC-V P extension specifications - GitHub*. URL: https://github.com/riscv/riscv-p-spec (cit. on pp. 72, 73).

[23] *RISC-V P Extension Proposal, SMAQA (Signed Multiply Four Bytes with 32-bit Adds) - GitHub*. URL: https://github.com/riscv/riscv-p-spec/blob/master/P-ext-proposal.adoc#smaqa-signed-multiply-four-bytes-with-32-bit-adds (cit. on p. 73).

[24] *Custom instruction in the software toolchain - GitLab*. URL: https://pcotret.gitlab.io/riscv-custom/sw_toolchain.html#adding-a-custom-instruction-in-the-cross-compiler (cit. on p. 74).