UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Jure Vreča

DEEP LEARNING ON LOW POWER EMBEDDED DEVICES USING RISC-V CORES WITH AN EXTENDED INSTRUCTION SET

Globoko učenje na nizkoenergijskih vgrajenih napravah z uporabo jeder RISC-V z razširjenim naborom ukazov

Master's thesis

Maribor, April 2020

UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Jure Vreča

DEEP LEARNING ON LOW POWER EMBEDDED DEVICES USING RISC-V CORES WITH AN EXTENDED INSTRUCTION SET

Globoko učenje na nizkoenergijskih vgrajenih napravah z uporabo jeder RISC-V z razširjenim naborom ukazov

Master's thesis

Maribor, April 2020

DEEP LEARNING ON LOW POWER EMBEDDED DEVICES USING RISC-V CORES WITH AN EXTENDED INSTRUCTION SET

Globoko učenje na nizkoenergijskih vgrajenih napravah z uporabo jeder RISC-V z razširjenim naborom ukazov

Master's thesis

Student:	Jure Vreča
Study program:	Master's study program of Electrical Engineering
Option:	Electronics
Mentor:	Prof. Dr. Zmago Brezočnik
Co-Mentor:	Dr. Farhad Merchant
	Assist. M.Sc. Ernest Gungl
Lector:	Rok Kostomaj, M.A.

Zahvala

Za pomoč pri izdelavi magistrske naloge bi se zahvalil mentorjema
prof. dr. Zmagu Brezočniku in dr. Rainerju Leupersu ter
somentorjema mag. Ernestu Gunglu in dr. Farhadu Merchantu.
Prav tako bi se rad zahvalil dr. Thomasu Grassu, gospodu Karlu Sturmu
in dr. Paolu Bientinesiu za nasvete pri izdelavi magistrske naloge.
Na koncu bi se rad zahvalil še staršem, ki so mi omogočili študij in me spodbujali, da ga končam.

Acknowledgment

For the help with the Master's thesis, I would like to thank my mentors,
Prof. Dr. Zmago Brezočnik and Prof. Dr. Rainer Leupers, and also my co-mentors
Asst. M.Sc. Ernest Gungl and Dr. Farhad Merchant.
I would also like to thank Dr. Thomas Grass, Mr. Karl Sturm, and
Dr. Paolo Bientinesi for help with the Master's thesis.
Additionally, I would like to express my gratitude to my parents, who gave me financial support and encouragement to finish my studies.

The work for this Master's thesis was done on the Institute for Communication Technologies and Embedded Systems of the RWTH Aachen University.

Globoko učenje na nizkoenergijskih vgrajenih napravah z uporabo jeder RISC-V z razširjenim naborom ukazov

Ključne besede: globoko učenje, vgrajeni sistemi, nabor ukazov, RISC-V
UDK: 004.8.021(043.2)

Povzetek

1. Uvod in motivacija

V magistrski nalogi smo raziskovali, kako različne razširitve nabora ukazov vplivajo na učinkovitost izvajanja algoritmov globokega učenja — predvsem v kontekstu nizkoenergijskih vgrajenih naprav. Algoritmi globokega učenja se redko uporabljajo v vgrajenih sistemih, ker so računsko in energijsko zelo zahtevni. Zatorej je bil stranski cilj magistrske naloge pokazati, da lahko algoritme globokega učenja v okrnjeni obliki kljub temu uporabljamo v nizkoenergijskih vgrajenih napravah. Naše delo temelji na [1]. V [1] avtorji predvidevajo, da lahko s krmiljenjem vezij blizu pragovnih napetosti in s pametnimi izboljšavami nabora ukazov dosežejo spodobno računsko moč pri zelo majhnni porabi energije.

2. Teoretično ozadje

V tem poglavju smo uvedli koncepte, ki jih mora bralec spoznati za učinkovito branje in razumevanje magistrske naloge. Začeli smo z analizo algoritmov globokega učenja. Delovanje teh algoritmov smo predstavili predvsem na računskem nivoju. Za tem smo pregledali delovanje vezij FPGA, na kratko tudi vodilo AMBA, operacijski sistem Linux in koncept odvijanja zank. Vodilo AMBA in Linux smo uporabljali za testiranje sistema. Koncept odvijanja zank pa je

optimizacija za prevajalnik, s katero smo skušali izboljšati delovanje algoritmov. Razložili smo, kaj je koncept nabora ukazov. RISC-V je odprt nabor ukazov, ki so ga pričeli razvijati na Univerzi v Kaliforniji, Berkeley, delo pa je nato nadaljevala fundacija RISC-V. RISC-V ima namen standardizirati nabore ukazov za vse vrste računalniških platform — od vgrajenih sistemov do namiznih računalnikov in strežnikov. Zato RISC-V ni samo en nabor ukazov, ampak so trije. Poleg le-teh pa obstaja še vrsta razširitev, ki dodajo specifične funkcionalnosti (na primer razširitev za aritmetiko s plavajočo vejico).

3. Strojna oprema

Za osnovo smo vzeli odprtokodni mikrokrmilnik PULPino, ki sta ga skupaj razvila ETH Zürich in Univerza v Bologni. Procesorsko jedro mikrokrmilnika PULPino se imenuje RI5CY. RI5CY je majhno jedro s cevovodom s štirimi stopnjami. Jedro je namensko zgrajeno za nizkoenergijsko paralelno računanje. Procesorsko jedro smo nato nadgradili z enoto za računanje skalarnega produkta vektorjev velikosti štiri. Vso aritmetiko smo delali v aritmetiki s plavajočo vejico. Pomembna funkcionalnost jedra RI5CY je funkcija strojne zanke. Ta funkcija nam je omogočala učinkovitejše izvajanje zank z majhnim telesom — t.j. z majhnim številom ukazov v telesu zanke.

4. Prototip z uporabo FPGA

Da smo lahko ocenili učinkovitost našega sistema, smo razvili testni sistem, ki je temeljil na Zynq-7000. Zynq-7000 je družina sistemov na čipu podjetja Xilinx. Čipi Zynq-7000 imajo na istem siliciju združeno tako trdo intelektualno lastnino kot tudi programirljivo polje logičnih vrat. Del trde intelektualne lastnine je procesor ARM Cortex-A9, na katerem smo poganjali aplikacijo v Linuxu, ki je sprogramirala, nadzirala in spremljala na FPGA-ju emuliran PULPino. Aplikacija je tudi izvajala meritve in zbirala rezultate. Procesor ARM Cortex-A9 in PULPino smo povezali z vodilom AXI. Po tem vodilu smo v PULPino dostavljali vhodne podatke za nevronsko mrežo. PULPino je o koncu računanja obvestil procesor ARM Cortex-A9 s prekinitvijo. Sistem smo načrtovali v programu Vivado podjetja Xilinx.

5. Nevronska mreža

V sklopu naloge smo razvili tudi preprosto demonstracijsko konvolucijsko nevronsko mrežo, ki smo jo poganjali na naši strojni opremi. Izvajala je optično zaznavo znakov z vsega petimi sloji — dva konvolucijska sloja, dva polno-povezana sloja in izhodni sloj tipa Softmax. Nevronsko *mrežo smo razvili s knjižnico TensorFlow podjetja Google. Prenos nevronske mreže na našo strojno opremo nam je delal težave, ker TensorFlow nima vgrajene podpore za generiranje kode za procesorje RISC-V. Da smo zaobšli ta problem, smo razvili knjižnico za globoko učenje za naš sistem v dveh različicah. Prva različica je bila referenčna verzija, ki je bila napisana v jeziku C in ni bila optimizirana. Druga različica je bila delno napisana v zbirniku in smo jo poskušali čim bolj optimizirati. Le druga različica je uporabljala našo dodano enoto za računanje skalarnega produkta vektorjev. Knjižnica za globoko učenje ima sedem funkcij.*

6. Analiza rezultatov

Ugotovili smo, da je dodana enota za računanje skalarnega produkta vektorjev povečala površino silicija za 172 %. Povečanje površine je več kot podvojilo statično porabo, medtem ko je dinamična poraba ostala na podobni ravni. Velikost naše knjižnice za globoko učenje se je glede na različico in izbrane optimizacije gibala med 1,2 KB in 4,5 KB. Ta velikost je sprejemljiva tudi za vgrajene sisteme, ki so praviloma zelo omejeni s pomnilnikom. Podali in analizirali smo rezultate za celotno demonstracijsko nevronsko mrežo in za različne tipe slojev posebej. Ugotovili smo, da je dodana enota za računanje skalarnega produkta bolj uspešna pri pospeševanju konvolucijskih kot polno-povezanih slojev. Glavni razlog za to je dejstvo, da so konvolucijski sloji manj pomnilniško požrešni — enoto za računanje skalarnega produkta smo lahko bolj izrabili. Konvolucijske sloje smo v najboljšem primeru izračunali z 78 % manj urinih ciklov, medtem ko smo polno-povezane sloje izračunali z 72 % manj urinih ciklov. Tako smo v povprečju zmanjšali število urinih ciklov za 73 %. Končni rezultat tega je bil, da se je energija, potrebna za izračun ene iteracije nevronske mreže, zmanjšala za 73 % v primerjavi z neoptimiziranimi različicami.

7. Primerjava z drugimi znanimi deli

Za konec smo naše delo primerjali z drugimi znanimi deli. Primerjali smo ga s pred kratkim izdano razširitvijo nabora ukazov ARM Helium in z [2]. Helium je podobna vektorska razširitev, namenjena za družino procesorjev ARM Cortex-M. Mikroprocesorji z razširitvijo ARM Helium še niso komercialno dobavljivi, zato je bila na voljo le peščica podatkov za primerjavo. V [2] so implementirali nekoliko bolj zapleteno razširitev nabora ukazov, vendar je niso testirali na dejanski strojni opremi, ampak le v simulatorju. Kljub temu smo lahko primerjali določene rezultate, kot je denimo število ukazov.

Deep Learning on Low Power Embedded Devices Using RISC-V Cores with an Extended Instruction Set

Key words: deep learning, embedded system, instruction set, RISC-V
UDK: 004.8.021(043.2)

Abstract

This thesis explores the possibility of running neural networks on microcontrollers and how to optimize their performance using instruction set extensions. Microcontrollers are seen as too weak to run neural networks. We challenge this view and show that stripped-down neural networks can run and be useful for some applications. We used an open-source microcontroller called PULPino to run our neural network. The benefit of various instructions and optimizations for minimizing energy consumption to run deep learning algorithms was evaluated. Hardware loops, loop unrolling, and the dot-product unit were implemented and tested.

We developed an FPGA-based testing system to evaluate our hardware. We also developed a deep learning library and a test neural network for our hardware. We wrote two versions of the deep learning library. One version is the reference code, and the other is the optimized code that uses the dot product unit. Using the testing system, we tested the performance of the two versions. The synthesis was run to determine the power and energy consumption. We also tried out various optimizations to see if the performance could be improved.

Using instruction set extensions and algorithmic optimizations we reduced the clock cycle

count by 72% for the convolutional layers and by 78% for fully-connected layers. This reduced power consumption by 73%. We compare our results with related research.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ACE	Advanced Coherency Extensions
AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application-specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CB	Connection Boxes
CISC	Complex Instruction Set Architecture
CLB	Configurable Logic Blocks
FPGA	Field-Programmable Gate Arrays
GPGPU	General-purpose Graphics Processing Unit
HDF	Hierarchical Data Format
HDL	Hardware Description Language
HLS	High-Level Synthesis
IOB	Input/Output Block
IoT	Internet of Things
IP	Intellectual Property
LUT	Lookup Table

ML	Machine Learning
NPU	Neural Processing Unit
OCR	Optical Character Recognition
PL	Programmable Logic
PS	Processor Subsystem
PULP	Parallel Ultra Low Power
RISC	Reduced Instruction Set Architecture
RNN	Recurrent Neural Network
RTL	Register Transfer Level
SB	Switch Boxes
SIMD	Single Instruction, Multiple Data
SoC	System-on-a-Chip
UIO	Userspace I/O

1.0 Introduction and motivation

In this chapter, we first talk briefly about deep learning algorithms and try to motivate their use on low power embedded devices. Next, we discuss what hardware solutions are available for running neural networks and compare their advantages and disadvantages. We try to motivate the use of widely available microcontrollers. Lastly, we discuss the content of this thesis and demonstrate some examples of neural networks being run on microcontrollers and other low power systems.

1.1 Deep learning on low power embedded devices

Deep learning is a family of machine learning (ML) algorithms based on various types of artificial neural networks. Machine learning is a subset of the field of artificial intelligence (AI). Artificial intelligence encompasses all techniques that try to mimic human intelligence. Artificial neural networks are algorithms inspired by the behavior of neurons in the human brain. Figure 1.1 illustrates the relationship between deep learning, machine learning, and artificial intelligence. The deep learning and neural network terms are sometimes used interchangeably. However, in general, neural network algorithms try to emulate the behavior of real neurons, while deep learning algorithms use simplified models of neurons to enable machines to learn.

Deep learning algorithms have been applied successfully to various applications, such as language translation, image recognition, speech recognition, recommender systems, and natural language processing.

However, many of these advancements have been limited to devices with enough resources to handle such computationally and memory-intensive tasks. There are many use cases where we would like to use such algorithms, but we cannot afford to do so. Many times, the reason is the high power consumption of the devices capable of running neural networks. One way to get around this problem is to offload this processing to the cloud. However, this solution is limited to areas with Internet access. This trend is sometimes also called AI at the edge.

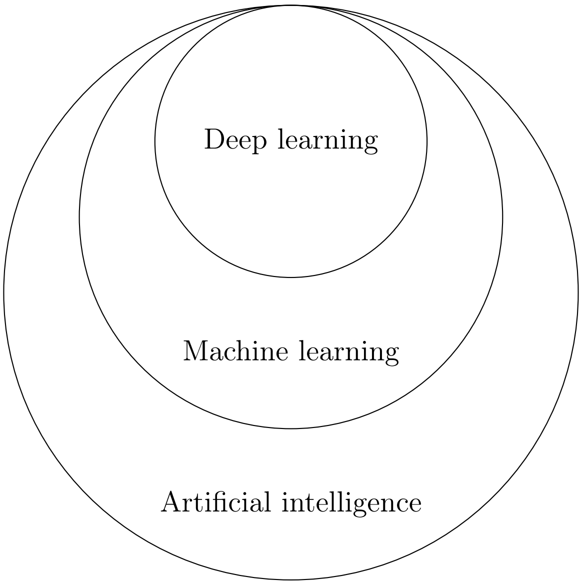


Figure 1.1: Artificial intelligence, machine learning, deep learning.

Limited Internet access is not the only reason for local processing. Other benefits of AI at the edge are also:

- low latency,
- privacy,
- reliability,
- security,
- energy-saving,
- the option of personalizing the algorithms for each device.

A lot of potential use cases of low power deep learning are in the Internet of Things (IoT) world and various autonomous robots. We list some potential use cases below:

- audio keyword spotting [5],
- human activity recognition (smartwatches) [5],
- facial recognition for surveillance cameras,
- autonomous vacuum robots,
- autonomous farming robots,
- autonomous drones,
- fall detection,
- sensor data fusion.

The applications above should mainly be considered when running on battery-operated devices. Designers of those devices have a trade-off to make between these features and the battery life of the product.

1.2 Available solutions

When considering the hardware of choice for running neural network inference, we must take into account several factors. The most important factors are:

- latency (the time it takes to compute the neural network),
- throughput (the number of the neural network runs that we can compute in a time unit),
- price,
- energy consumption.

The typical hardware used for deep learning is the general-purpose graphics processing unit (GPGPU). Most GPGPU hardware is designed for the personal computer market. In the low power embedded world, GPGPUs are not used because of power and price constraints. Another potential issue with GPGPUs is that they perform a technique called batching to achieve higher throughput. This technique increases latency, which might be a problem in certain applications.

Instead, various domain-specific architectures have been proposed to fill this gap because deep learning algorithms are highly parallelizable and have some other properties that specialized architectures can better exploit. The architectures mentioned above are sometimes referred to as neural processing units (NPU). NPUs have the potential to offer the best performance. One drawback of the NPU could be the relatively high price that comes with all new architectures. Also, because deep learning is a fast-changing field, it could happen that future algorithms do not work on such hardware or offer poor performance. The list below gives some existing commercial and non-commercial low power NPU designs:

- Eyeriss v2 [6],
- Efficient Inference Engine (EIE) [7],
- Texas Instruments Embedded Vision Engines (EVE) [8],
- ARM Machine Learning Processor [9],
- GreenWaves Technologies GAP-8: A RISC-V SoC [10],
- XNOR Neural Engine [11],
- Think Silicon NEMA| xNN,
- Google Edge TPU.

An alternative to using NPUs is using field-programmable gate arrays (FPGA). Even though

FPGAs can never offer the same performance as NPUs, they do offer the possibility of reconfiguration. Another advantage of FPGAs is that they already come in a variety of package sizes and with a varying number of lookup tables (LUT). They are thus enabling the embedded device designer to choose the appropriate size for a given problem.

Neural networks can also run on general-purpose processors. They come in a variety of sizes, from 8-bit microcontrollers to application processors. They are also easily programmed for any application. The problem with processors is their relatively low throughput if you are not running on a high-performance processor. However, if you do have such a processor, then the power consumption of it and its price is likely a problem for many embedded devices.

Companies from the embedded world have started to show some support for the topic. Table 1.1 lists some of the available hardware and software solutions.

1.3 Neural networks on microcontrollers

Microcontrollers are cheap systems on a chip found in almost every new product. IoT Analytics [12] reports that the number of IoT devices will grow to 9.9 billion by 2020. All of them must have a sort of processor, typically a microcontroller. If such microcontrollers would be able to run neural networks, a broad range of devices could offer smart features.

However, microcontrollers were typically seen as too week to run neural networks. Recent advancements in compression of neural networks [13] and modern neural network architecture [14] have opened up new possibilities. If we have neural networks small enough, we can minimize the high energy consumption associated with large memories such as DRAM. Figure 1.2 shows the relative energy consumption of accessing the memory of various sizes.

We believe that for specific low-intensity applications, microcontroller class processors with properly optimized instruction sets can provide a satisfactory solution. However, this is only the case if the neural networks can fit into the typically quite limited on-chip memories of microcontrollers. As can be seen from Figure 1.2, the price of adding an external memory like DRAM is an order of magnitude more energy expensive.

Table 1.1: AI solutions aimed at the low power embedded market.

Company	Product/Service	Short explanation
ARM	CMSIS-NN	Neural networks software library for Cortex-M processor cores.
ARM	ML Processor	Processor optimized for running deep learning algorithms.
ARM	Helium	Vector extension for Cortex-M processors aimed at speeding up signal processing and deep learning applications.
STMicroelectronics	STM32Cube.AI	Ecosystem for easily porting neural networks to STM32 microcontrollers.
Texas Instruments	Embedded Vision Engine (EVE)	Specialized vector processor aimed at ADAS applications.
Lattice Semiconductor	sensAI	Framework for bringing neural network algorithms to Lattice FPGAs.
Xilinx Inc.	AI Edge Platform	Set of tools and models for bringing neural network deployment on Xilinx FPGAs.
Google LLC	TensorFlow Lite for Microcontrollers	Experimental port of TensorFlow Lite aimed at microcontrollers.
Au-Zone Technologies Inc.	DeepView 2.0 ML Toolkit	Development tools to help embedded designers train and deploy neural networks.
Audio Analytic	ai3™	Prebuild neural networks for embedded consumer technology. They allow speech and music recognition.
MathWorks	MATLAB	Deep learning toolbox and C++ code generator for embedded deployment environments.
Pilot AI	Platform	Deep learning-based computer vision solutions that run directly on commodity embedded platforms.

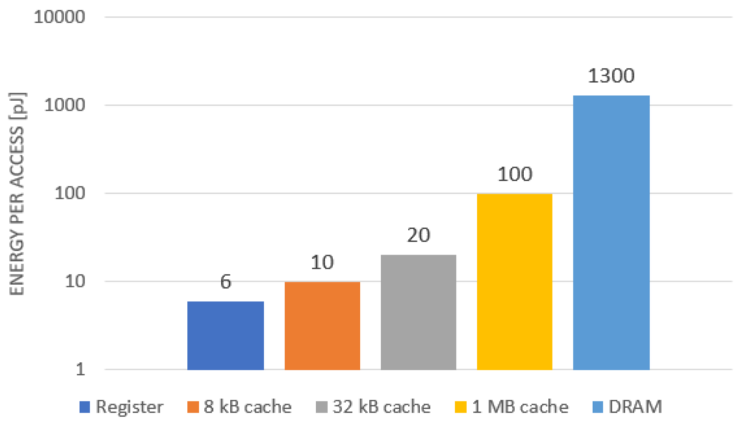


Figure 1.2: Energy consumption of various memories [15].

ARM, the dominant microprocessor designer for the embedded market, has recently released their single instruction, multiple data (SIMD) extension for their Cortex-M processors called Helium [4]. Helium is meant to accelerate signal processing and deep learning applications, but as of now, no implementations are supporting this extension[1]. Similarly, the RISC-V has a vector extension. In [16] they discuss the possibility of using such an extension for deep learning.

1.4 Examples

To motivate the reader, we list some existing examples of neural networks being run successfully on microcontrollers.

In [17] they developed a real-time fall detection system for wearable devices based on deep learning. They addressed the issue of unintentional falls among the elderly. Such falls are the leading cause of fatal and non-fatal injuries among the elderly [18]. They feed the data of tri-axial accelerometers to a recurrent neural network (RNN). If the RNN detects a fall, the system can issue a remote notification. The system was able to detect 98% of the falls in the SisFall dataset. The device battery lasts around 20 hours without recharging. They implemented this system on a low power ARM Cortex-M based microcontroller from STMicroelectronics. The entire board is 13.5×13.5 mm in size.

Similarly, in [19] they propose an early seizure detection system based on convolutional neural networks running on a microcontroller embedded in the body. The system measures electroencephalography (EEG) data and feeds it to the neural network, which determines the

[1] ARM Helium is not to be confused with NEON extensions. NEON is also a SIMD extension, but it is for the more powerful Cortex-A microprocessors.

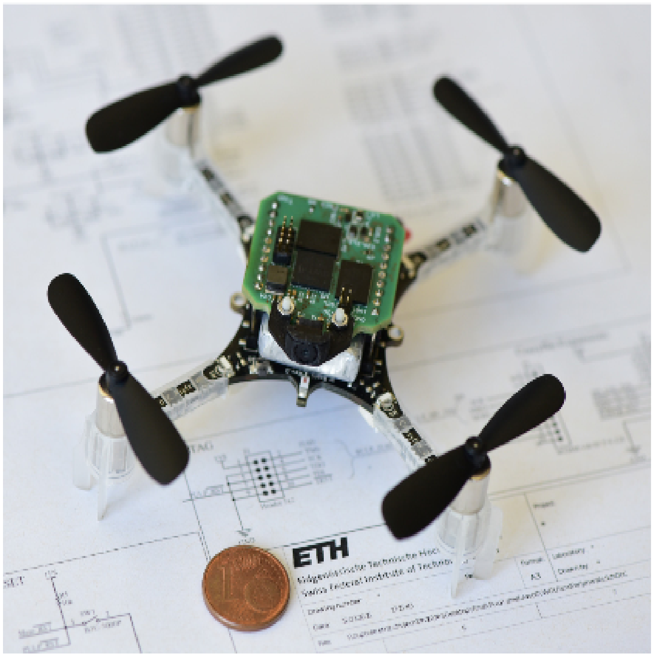


Figure 1.3: Prototype of the autonomous miniature drone [21].

seizure activity. They implemented the neural network on a low power microcontroller from Texas instruments.

In [20] they show that detection of keywords[2] in human speech can be done in real-time using microcontrollers. Keyword spotting systems are used to activate more elaborate speech recognition systems. That is because always-on speech recognition is not energy efficient. It also raises some privacy concerns if audio data is continuously streamed to the cloud. They explored different neural network architectures and optimized their memory footprint without significant cuts in prediction accuracy. The network achieved a 95.4% accuracy with only a 70 KB memory footprint. This example was also tested on an ARM Cortex-M based microcontroller from STMicroelectronics. The code, model definitions, and pre-trained model are available online [20].

In [21], they implemented a fully-autonomous miniaturized drone with visual navigation capabilities based on artificial intelligence. Their system uses GAP8. It is a commercial low power computing platform. The drone measures only 92 mm from side to side. The system can run DroNet at 6 to 18 fps and use it for indoor and outdoor navigation. It is capable of around 7 minutes of flight. The drone's design is fully open-source. Figure 1.3 shows the prototype of the miniature drone. The drone is based on the open-source COTS Crazyflie 2.0 nano-quadrotor design.

[2]For example "Alexa", "ok Google", "Hey Siri", etc.

Summary

We started this chapter by presenting what is deep learning and discussing some hardware solutions for neural networks. We then presented a case for running small scale neural networks on widely available low power microcontrollers. We concluded by showing some examples of neural networks on microcontrollers. Next, we take a look at some concepts behind the algorithms and hardware we use in this thesis.

2.0 Background and concepts

In this chapter, we introduce some basic concepts that we need in this thesis. We start by discussing neural networks from an algorithmic point of view. We then move to hardware and software solutions we use to test our system. Finally, we discuss the RISC-V instruction set architecture, which we expand with our instructions.

2.1 Algorithm analysis

This chapter outlines the mathematical behavior of the algorithms that are the topic of this thesis. Neural network algorithms are loosely inspired by the behavior of neurons in the human brain, but they are not the same. These types of algorithms are also a bit different than standard algorithms; they first have to be trained, and then one can use them. The training process is the process of determining the weights of the network. After that, the action of running the obtained algorithm has different names: inference, prediction, evaluation, running, scoring, or testing. We will not concern ourselves with training the network; our focus is merely an inference.

The following subsections will introduce some basic concepts of neural networks.

Figure 2.1 shows a simplified example of a human neuron. As artificial neurons, they have inputs and outputs. The main difference lies in the fact that the relationship between inputs and outputs is very complicated and based on chemical processes. Neurons in the human brain have been trained over many millennia by evolution. Our artificial networks are sometimes close to human-level ability but solely in particular tasks. We are far away from emulating general intelligence, such as human intelligence.

2.1.1 Artificial neurons

As mentioned above, neural networks are called so due to being inspired by the behavior of human brain cells. However, the model of neurons used in artificial neural networks is a

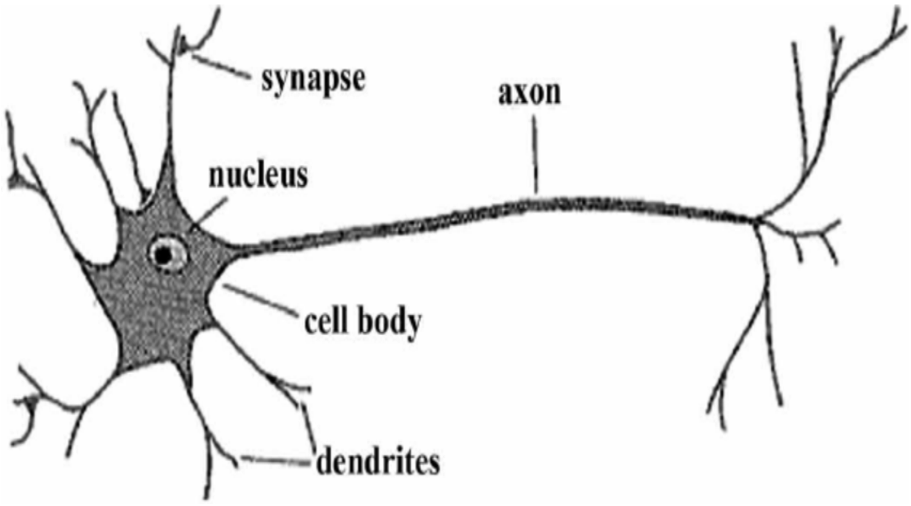


Figure 2.1: A simplified visualization of a neuron cell found in the human brain [22].

highly simplified model of actual brain cells. Figure 2.2 shows a symbolic representation of an artificial neuron. Akin to real neuron cells, it has inputs (x_i) and an output (y).

A single artificial neuron is an elementary unit from which artificial neural networks are constructed.

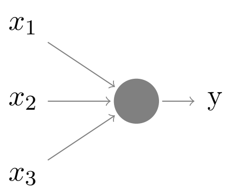


Figure 2.2: An artificial neuron.

The output y of this neuron is computed in the following way:

$$y = f(b + x_1w_1 + x_2w_2 + x_3w_3). \tag{2.1}$$

Or more generally, because the number of inputs can vary, we can write

$$y = f(b + \sum_i x_iw_i) \tag{2.2}$$

where w_i and b are constants called the weights and the bias, respectively. x_i terms are the inputs to the neuron, y is the output of the neuron, and f is a nonlinear function. There are several choices for the nonlinear function f. Most commonly used are Sigmoid, tanh and rectified linear unit (ReLU).

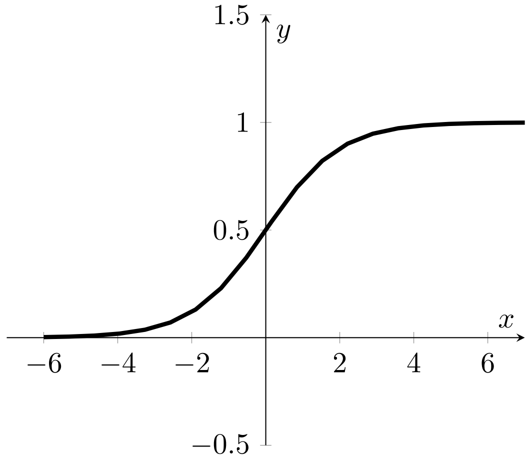


Figure 2.3: Plot of the Sigmoid function.

The Sigmoid function is defined as

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{2.3}$$

the hyperbolic tangent (tanh) function as

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{2.4}$$

and the ReLU function as

$$f(x) = max(0, x). \tag{2.5}$$

Figure 2.3, Figure 2.4, and Figure 2.5 show the plots of these nonlinear functions. The Sigmoid and the hyperbolic tangent function are computationally much more expensive than the ReLU function. This fact is the reason that most of the newer neural networks use the ReLU function. Other functions have also been investigated but have not found wide adoption. Note that the function applied to the output of the artificial neuron must be nonlinear. If it is not, the neural network will not be able to learn much.

We need more than just one neuron to make something useful. So next, we will look at how we connect many of these neurons to form more complex structures.

2.1.2 Layers of neural networks

Neural networks are layered. Every neural network has an input layer and an output layer. Every layer in between the input layer and the output layer is called the hidden layer. Modern

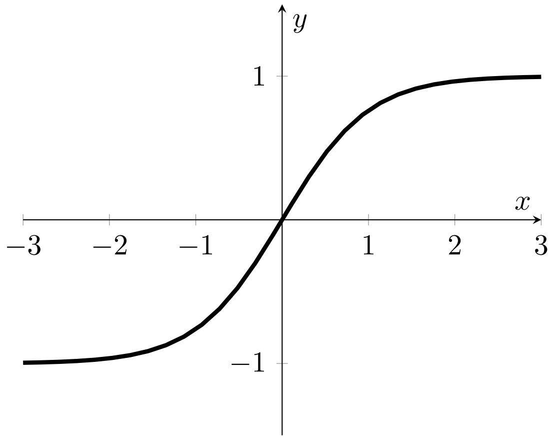


Figure 2.4: Plot of the tanh function.

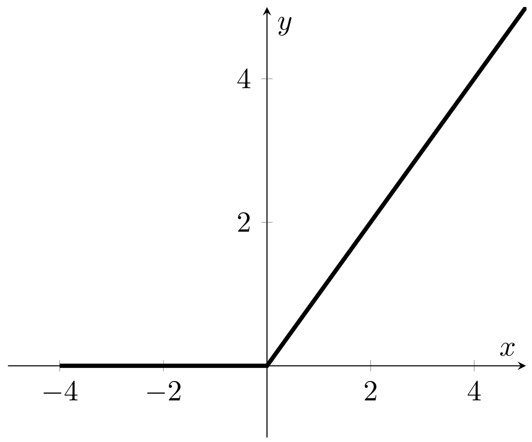


Figure 2.5: Plot of the ReLU function.

networks have many hidden layers. The term ”deep learning” comes right from this fact. Generally speaking, the deeper the network is, the more accurate it is.

The most prevalent layer types are the fully connected layer, the convolutional layer, the pooling layer, and the softmax. They will be presented in the ensuing part.

2.1.2.1 Fully connected layers

Fully connected layers are layers in which all neurons of the previous layer are connected to all neurons of the next layer. Dense neural networks are comprised of only such layers only. An example of such a network is shown in Figure 2.9. It has an input layer, a single hidden layer, and an output layer.

It has an input layer, a single hidden layer, and an output layer. Computationally speaking, fully connected layers are computed as a matrix-vector multiplication and applying the nonlinear function. Equation (2.6) shows the mathematical operation that is being performed to compute one fully connected layer.

$$y = f(W \cdot x + b) \tag{2.6}$$

W $[M \times N]$ is the matrix of weights of the layer, x $[N \times 1]$ is the input vector to the layer, b $[M \times 1]$ is the bias term vector, f is the nonlinear function, and y $[M \times 1]$ is the output vector. M is the number of neurons in the next layer and N is the number of neurons in the previous layer or the number of inputs to the neural network.

To illustrate this further, let us take the network in Figure 2.9 as an example. Let us presume, we already trained the network, and so the coefficients are known to us. Every line in Figure 2.9 represents a coefficient. We will label them using the names and indexes of the layers they connect. For example, the name of the neurons in the Input layer are I_1, I_2, I_3, I_4, the name of the neurons in the Hidden layer are H_1, H_2, H_3, H_4, H_5 and the name of the output layer neuron is O_1. Using this scheme, we label a connection between I_1 and H_5 as IH_{15}. This connection is highlighted in red in Figure 2.9. Accordingly, we name the bias terms BH_1, BH_2, BH_3, BH_4, BH_5, and BO_1.

Using this naming scheme, Equation (2.7) shows how we can calculate the output of the neural network in Figure 2.9. In this case, because we have a single output neuron, the output is scalar. Generally, the output is a vector.

$$Output = f\left(\begin{bmatrix} HO_{11} & HO_{12} & HO_{13} & HO_{14} & HO_{15} \end{bmatrix} \cdot f\left(\begin{bmatrix} IH_{11} & IH_{21} & IH_{31} & IH_{41} \\ IH_{12} & IH_{22} & IH_{32} & IH_{42} \\ IH_{13} & IH_{23} & IH_{33} & IH_{43} \\ IH_{14} & IH_{24} & IH_{34} & IH_{44} \\ IH_{15} & IH_{25} & IH_{35} & IH_{45} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} BH_1 \\ BH_2 \\ BH_3 \\ BH_4 \\ BH_5 \end{bmatrix}\right) + BO_1\right) \tag{2.7}$$

Note that the nonlinear function is applied to every element of the vector separately.

2.1.2.2 Convolutional layers

Unlike the fully connected layer in convolutional layers, not every neuron is connected to each neuron of the next layer. Instead, each neuron from the next layer is connected to a bundle of neuron in the previous layer. Convolutional layers are called so because the operation they are performing resembles the operation of 2D Convolution. The only difference is that we do not reflect the signals. Such an operation is known in signal processing as 2D cross-correlation.

The operation is shown in Figure 2.6 and Algorithm 2.1. The convolution is performed between an input image I and a two-dimensional array of weights named a kernel K. Kernels usually have small dimensions, e.g., 5×5, 3×3, or even 1×1. We slide the kernel over the image and compute the dot product between I and K. This produces a so-called activation map. The size of the step when moving the filter is called the stride (it need not be equal to one).

For example, to compute the bottom right result, we multiply the corresponding elements in the red highlighted part of I and K. This procedure is shown in Equation (2.8). The final integer result is obtained by adding all the elements of the rightmost array (in our case, the result is 8).

$$
\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} -> \begin{bmatrix} 1\cdot 1 & 1\cdot 1 & 0\cdot 1 & 0\cdot 1 & 1\cdot 1 \\ 1\cdot 1 & 0\cdot 0 & 0\cdot 0 & 0\cdot 0 & 0\cdot 1 \\ 0\cdot 1 & 0\cdot 0 & 0\cdot 1 & 0\cdot 0 & 1\cdot 1 \\ 0\cdot 1 & 0\cdot 0 & 0\cdot 0 & 0\cdot 0 & 0\cdot 1 \\ 1\cdot 1 & 0\cdot 1 & 0\cdot 1 & 1\cdot 1 & 1\cdot 1 \end{bmatrix} -> \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.8}
$$

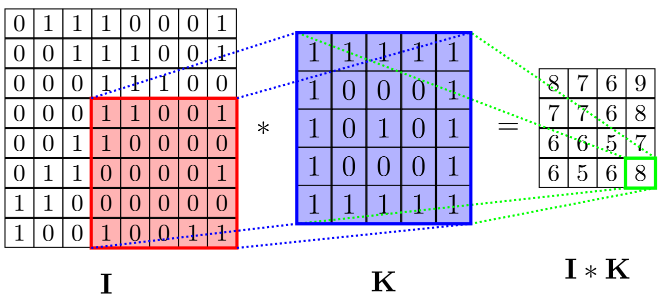


Figure 2.6: The convolution operation.

The convolutional layers change the dimension of the input. Sometimes this is desired, but at other times it is not. Should we wish to retain our dimensions, we may add zero-padded borders to the input image.

2.1.2.3 Pooling layers

Pooling layers are typically used in conjunction with convolutional layers. They are used to reduce the spatial dimension. In effect, they are summing up the results of the previous layer by down-sampling the input. Pooling layers take a look at the input array in smaller chunks.

Algorithm 2.1: The convolution operation.

input : image *img* of size $img\_size \times img\_size$
input : filter *fil* of size $fil\_size \times fil\_size$
input : stride *stride*
input : bias *bias* of size 1
output : result *res* of size $out\_size \times out\_size$

```
out_size = ((img_size - fil_size)/stride) + 1;
// i and j get incremented by stride after each iteration
for i = 0 to out_size - 1 do
    for j = 0 to out_size - 1 do
        img_slice ← img[i][j]; //img_slice is a subset of img
        res[i][j] = bias;
        for k = 0 to fil_size - 1 do
            for l = 0 to fil_size - 1 do
                res[i][j] = res[i][j] + img_slice[k][l] × fil[k][l];
```

The sizes of these chunks are usually 2×2 or 3×3. On that chunk, they perform a specific operation. For example, the maximum pooling operation takes the highest number of the chunk. This operation is best explained by an example shown in Figure 2.7. Another option is average pooling, which, as the name suggests, calculates the average of the numbers.

input

99	11	22	20
55	12	20	33
11	25	55	55
10	33	55	13

output

99	33
33	55

Figure 2.7: The maximum pooling layer.

Because pooling layers significantly reduce the dimensions, the layers following a pooling layer are much less computationally expensive.

2.1.2.4 Softmax

The softmax function is typically used at the very end of the network, which has more than one output. It limits all outputs between zero and one. The softmax takes in the output vector of the network and outputs a vector of the same size. The result of the softmax function can be interpreted as a probability that a specific output is correct. In other words, the sum of the output vector is one. Equation (2.9) shows how to compute the i^{th} output O_i, where a is the input vector of size N, O is the output vector of size N, and β is a normalization coefficient.

$$O_i = \frac{e^{\beta \cdot a_i}}{\sum_j^N e^{\beta \cdot a_j}} \tag{2.9}$$

The softmax function is for large output vectors computationally fairly expensive. For this reason, it is rarely used if the neural network has many output neurons.

Before softmax		After softmax
2.0		0.61
1.2	$\longrightarrow$	0.28
0.3		0.11

Figure 2.8: An example of the effect of the softmax function.

Figure 2.8 shows the effect of using a softmax function on an output vector for $\beta = 1$. Equation (2.10) details how to compute the first parameter of the example in Figure 2.8.

$$\frac{e^{1 \cdot 2.0}}{e^{1 \cdot 2.0} + e^{1 \cdot 1.2} + e^{1 \cdot 0.3}} = 0.61 \tag{2.10}$$

By choosing the parameter β, we can tweak the output of the softmax functions so that it is more or less sensitive to the differences in the input.

2.1.3 Neural network architecture

Neural network architecture is defined by the types of layers we use. For example, a network consisting of only fully connected layers is called a dense neural network. Here is a list of some commonly used architectures:

- dense neural network,
- convolutional network,
- single-layer perceptron,
- recurrent neural network,
- long/short time memory,
- multiplicative interactions,
- transformer networks.

Some architectures work better for some problems than others. For example, convolutional neural networks work better for image processing, but recurrent neural networks are better

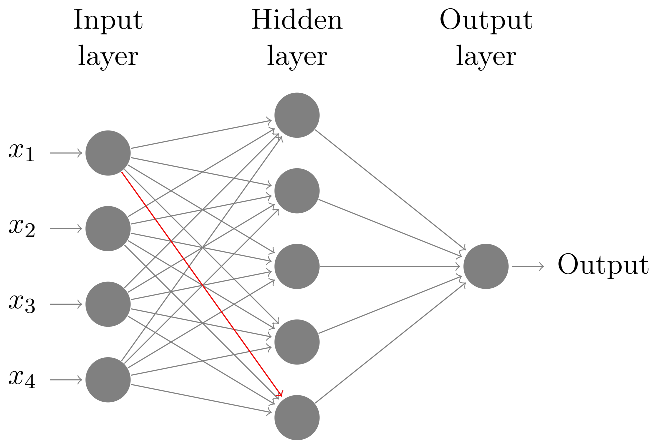


Figure 2.9: Example of a dense neural network.

for speech recognition. Figuring out which neural network architecture works better for a particular task is mostly based on experience.

Modern neural networks have many layers. Figure 2.10 shows the architecture of a convolutional neural network called AlexNet. It consists of 650,000 neurons and 60 million parameters.

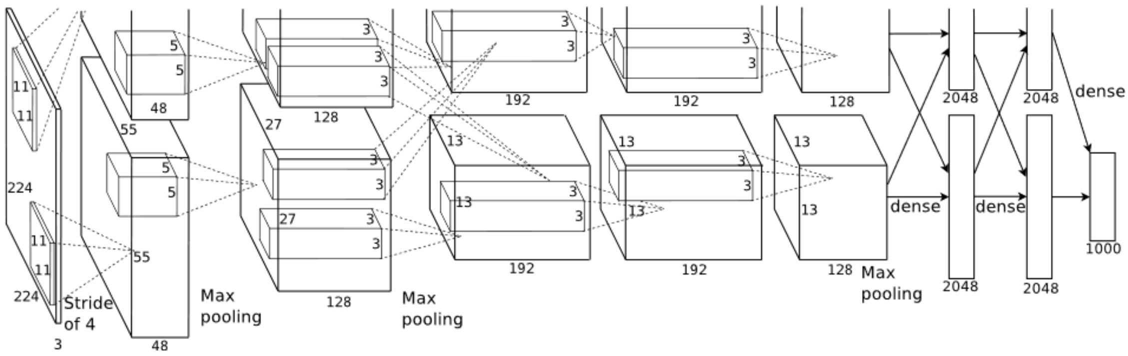


Figure 2.10: Architecture of AlexNet [23].

2.2 Field-programmable gate array

FPGA is a class of integrated circuits that can be configured after fabrication. FPGAs contain an array of configurable logic blocks (CLB) wired together by configurable interconnects. A CLB is sometimes also called a programmable logic block. By connecting the logic blocks, FPGAs can implement both combinational and sequential logic. The combinational logic is achieved by configuring small memory cells called lookup tables (LUT). By adding a D flip-flop to that, we get sequential behavior.

A CLB is made up of two to eight logic cells or slices. The exact details differ depending on the manufacturer, but generally, the structure of a slice is shown in Figure 2.11. The latest high-end FPGA devices incorporate up to 5 million logic cells [24].

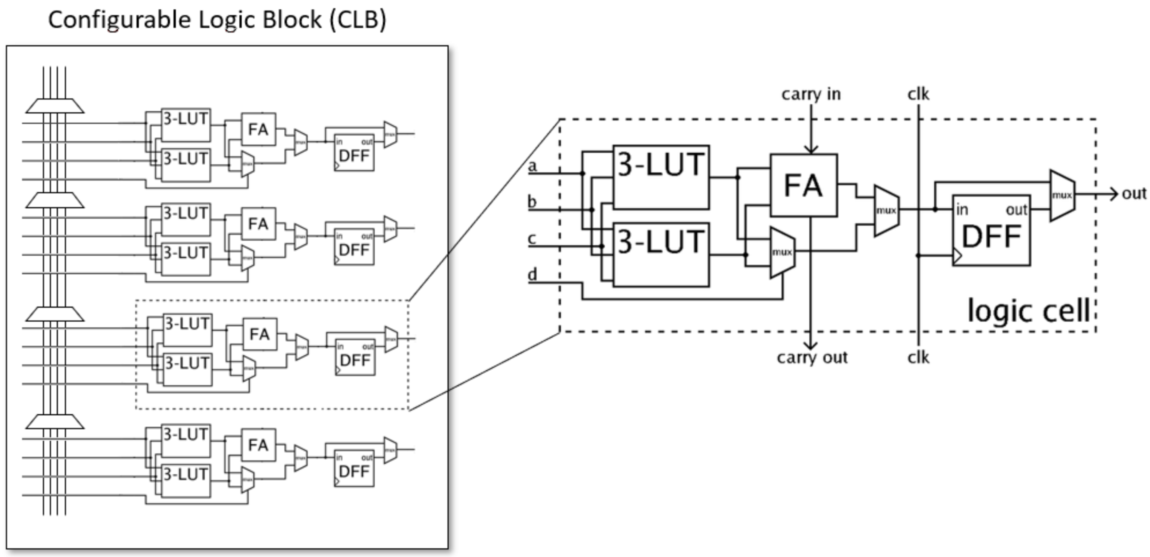


Figure 2.11: A simplified block diagram of a CLB and a logic cell [25].

CLBs are wired together by an interconnect structure consisting of wires, connect boxes (CB) and switch boxes (SB). CBs and SBs contain an assortment of transistors that enable us to make and break connections between interconnect wires. An example of an FPGA structure is shown in Figure 2.12. FPGAs also contain other blocks with dedicated functionality such as

- input/output blocks (IOB),
- clock management blocks,
- dedicated RAM blocks (BRAM),
- dedicated arithmetic units.

These additional blocks are there to perform specific tasks that are common in many designs. They improve FPGA performance. For example, memory can be generated using BRAM or LUTs. In most cases, we should use BRAM to save on the amount of LUTs we use.

2.2.1 FPGA design tools

As in standard integrated circuit design, we use hardware description languages (HDL) to determine the behavior of the FPGA. The most common languages are VHDL and Verilog[1]. They are both register transfer level (RTL) languages. Logic synthesis is the process of interpreting the abstract RTL specification into a circuit of logic gates. Logic synthesis is

[1] There exists an extension to Verilog called SystemVerilog that is also in common use.

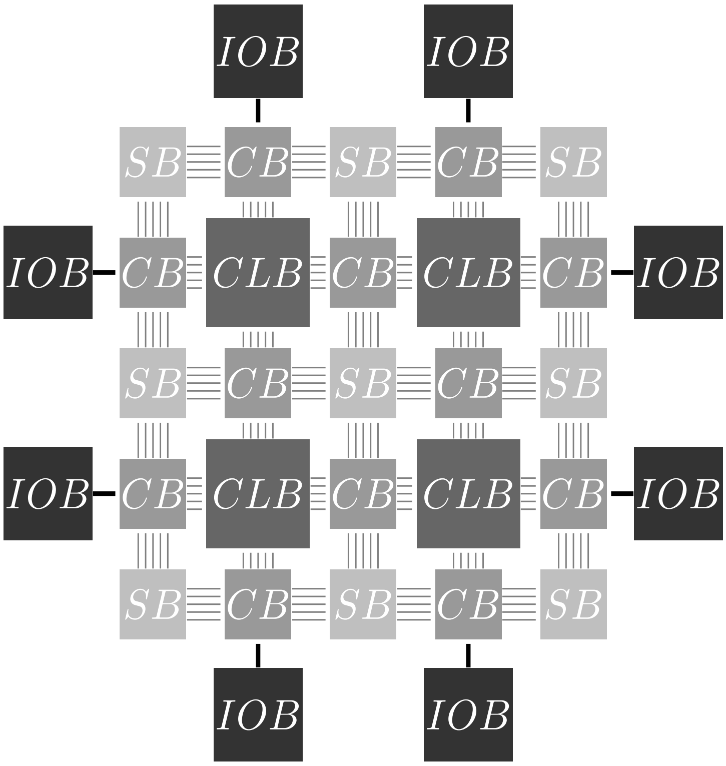


Figure 2.12: A simplified block diagram of a typical FPGA device.

performed by relatively complicated and often expensive software.

There also exist design methods with a higher-level abstraction. High-level synthesis (HLS) increases the abstraction level and enables faster hardware development [26]. HLS tools typically convert high-level languages to Verilog or VHDL, which can then be passed to conventional synthesis tools.

Typically, these tools also offer extensive debugging and verification capabilities. Verifying if the design works properly, is a significant task. Hardware description languages offer various language constructs that are only intended for verifying if the design is functioning as expected. That means that they cannot be synthesized to produce an actual physical circuit. Instead, they are only usable in a simulated environment.

We used Vivado Design Suite from Xilinx. It is a major FPGA manufacturer, and they provide their electronic design automation tools at a meager cost. A version of Vivado is free for non-commercial use. Vivado additionally has features for system-on-a-chip (SoC) development and a rich set of configurable intellectual property (IP) cores. The configurable IP cores enable a designer to make new designs quickly.

2.2.2 Intellectual property cores

Intellectual property cores are reusable logic designs. They can be anything — for example, processors, memory controllers, communication modules etc. The IP design philosophy is not to reinvent the wheel every time you start a design. Instead, you can make your design by connecting different IP cores and configuring them appropriately.

Intellectual property cores can either be hard IP cores or soft IP cores. Soft cores are typically offered as synthesizable RTL — so either in Verilog or VHDL files. Hard IP cores are not modifiable and are provided in a low-level representation (for example, post layout). Hard IP provides better results, but cannot be used in a purely FPGA design. Xilinx provides a wide range of different IPs. It provides processor cores, video compression cores, interface cores like USB controllers, and many others. All these IP cores come prebuilt into the Vivado Design Suite, but other vendors can also offer IP through the Vivado environment.

As mentioned before, we must connect the cores in some way to make them usable together. The most common configuration for connecting several IP cores is the bus configuration. The IP cores provided by Xilinx mostly use the open-source Advanced Microcontroller Bus Architecture (AMBA) interfaces. AMBA is explained in the next section.

2.3 Advanced Microcontroller Bus Architecture

AMBA is an on-chip interconnect specification for the connection and management of IP cores in the system on a chip (SoC) designs. AMBA is an open-standard from ARM and can be used without royalties[2]. AMBA was introduced in 1996, but since then, several versions of AMBA have been released. The current specification is the fifth version. We will discuss AMBA version four because it is the one used in our system. Newer versions generally do not change the standard but extend it.

It is essential to have an open, interconnect standard to encourage design reuse. Having a common interconnect standard ensures that the various IP blocks can function together.

Despite its name, AMBA is useful for more than just microcontroller devices. Today's ASIC and SoC devices widely use AMBA, including also many application-level processors used in modern smartphones and even some laptops. For embedded processors, AMBA has become a de-facto.

The AMBA 4 specification specifies the following bus protocols:

[2] There are some restrictions on the use of AMBA. For details, look at the license on the ARM's website.

- the Advanced eXtensible Interface (AXI) and the Advanced Coherency Extensions (ACE),
- the AXI4-Lite specification,
- the Advance Trace Bus specification,
- the Advanced Peripheral Bus (APB) specification,
- the Low Power Interface specification.

Various use cases require different buses. We will look at the AXI bus and the APB bus since it is these two that we will use in our design.

2.3.1 The Advanced eXtensible Interface (AXI)

The AMBA Advanced eXtensible Interface (AXI) protocol is an on-chip protocol for high-performance, high-frequency designs. The AXI protocol also includes optional extensions for low power operation. The key features of AXI are [27]:

- separate address/control and data phases,
- it has support for unaligned data transfers,
- it has support for burst transactions,
- separate read and write data channels,
- support for out-of-order transaction completion,
- permits easy pipelining[3].

The AXI protocol also has the Advanced Coherency Extensions (ACE). They are used to provide synchronization in multiprocessor configurations.

There exists a deteriorated version of the AXI bus with the Advanced Coherency Extensions. It is called ACE-Lite. The Accelerated Coherency Port or ACP is a port defined for use with the ACE-Lite protocol.

Figure 2.13 and Figure 2.14 show the transaction diagrams for write and read transactions on an AXI bus. As can be seen from the figures, AXI uses both a burst read transaction and burst write transactions. In both cases, only the start address is issued.

[3] Pipelining is a procedure where you insert register stages into the combinational logic. If done right, this increases the maximum operating frequency of the circuit.

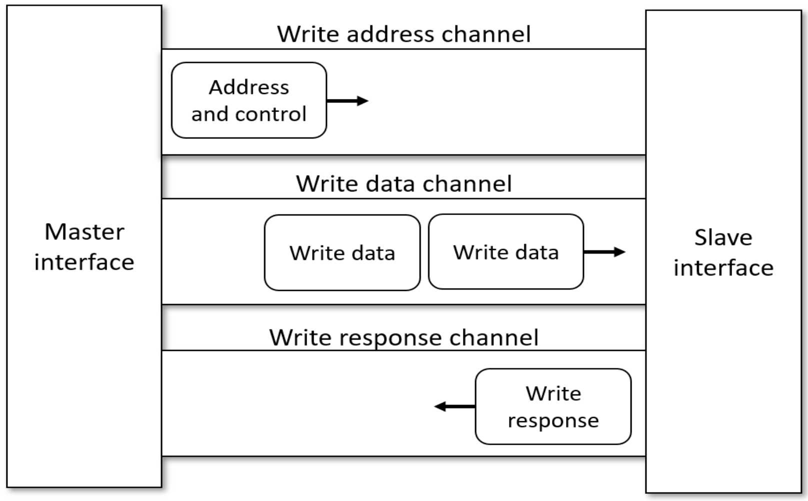


Figure 2.13: Channel architecture of writes.

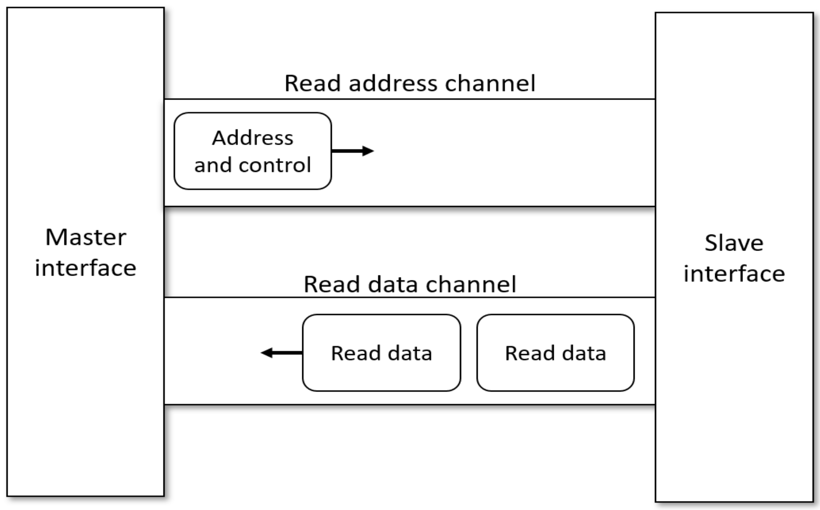


Figure 2.14: Channel architecture of reads.

2.3.2 The Advanced Peripheral Bus (APB)

The Advanced Peripheral Bus (APB) is designed as a low cost, low bandwidth bus that is optimized for minimal power consumption [28]. The APB protocol does not allow pipelining, and thus, it is slow. It should only be used to access programmable control registers or to connect slow devices such as peripherals (hence the name). APB was defined already in the first version of AMBA back in 1996.

The APB bus has only two channels. One for reading data and one for writing data. However, both channels can not be used simultaneously because they do not have separate handshake signals. The buses can be up to 32 bits wide.

2.4 Linux operating system

Linux is an open-source operating system initially written by Linus Torvalds in 1991. An operating system is a piece of software that acts as an abstraction layer for other applications. It handles access to memory, peripherals, and other resources. Linux typically comes in something called a Linux distribution. It is a combination of the Linux kernel and supporting system software. Linux is written in a way that it is hardware independent. It means that it can run on many different architectures, including the x86, RISC-V, and ARM.

Linux is the leading operating system on servers and supercomputers. It is used as an operating system for many personal computers and is also very widely used on embedded devices.

2.4.1 Linux kernel basics

The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system [29]. A process is an executing program that uses the kernel to provide it with basic functionality such as writing to screen, accessing memory, getting interrupts etc. The kernel enables many processes to have access to the same hardware. This is achieved on a single core by sequentially changing which process is to be being run. It is the job of the scheduler to determine which process should be running and which waiting. Application code running on Linux can be written as if it is the only application running on the processor. To prevent application software from interacting directly with low-level hardware components, the processor must support at least two different execution modes: a privileged mode and a non-privileged mode. In Unix terminology, this is called the user mode and kernel mode [30]. The kernel runs in kernel mode, and the other applications run in user mode. To access the hardware, a user mode application must invoke the kernel using a system call.

Linux is written in a modular fashion. The microkernel is the minimum amount of software required to implement an operating system. It constitutes basic functionality, such as process and memory management. All other modules can be either statically linked in at compile time or loaded in as needed. Such an approach makes the Linux kernel highly flexible. It is also the reason Linux can be used both for embedded applications and as a full operating system for workstations. The Linux binary can be as small as 12 MB [31].

2.4.2 Device trees

Instead of having the hardware specifications detailed in code, the Linux kernel uses something called a device tree to specify the hardware available on the system. Previously, Linux used to have a vast number of board-specific C source files. This fact was a problem mainly for embedded devices that are placed on many different boards [32].

A device tree is a tree structure used to describe the physical hardware in a system. The tree consists of nodes which represent characteristics of the represented devices. The device tree also contains fields that relate devices with physical addresses. That way, the address map is also specified with the device tree.

To write a device tree, one must follow the device tree specification made by the Device Tree organization [33]. The device tree is written in text form. After that, you must use a tool called a device tree compiler. This tool outputs a binary file called a device tree blob, that is what Linux then uses to identify the available hardware. Listing 2.1 is an example of a simple device tree. It defines two processing units and a serial communication module. Every node in the tree requires the compatible keyword. It is the key that Linux uses to decide which device driver to bind to a device.

Listing 2.1: A basic device tree [3].

```
/dts-v1/;
/ {
    compatible = "acme,coyotes-revenge";

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };

    serial@101F0000 {
        compatible = "arm,pl011";
    };
};
```

2.4.3 Device drivers and The Userspace I/O

Since each hardware that we attach to a computer may have its specific functionality, we need specific code to interact with that hardware. A software called device driver plays this role. Because device drivers interact with the hardware, they have to run in kernel mode. That means they are part of the kernel. The user space software uses a programming interface to interact with the device drivers, which then interfaces the actual hardware. Device drivers are part of the kernel.

Linux offers generic device drivers for simple hardware. They are offered by The Userspace I/O (UIO) system because, for many types of devices, creating a custom device driver is not needed. What is necessary is just access to the memory space of the device and some way to handle an interrupt [34]. A UIO device must be specified in the device tree.

The apparent advantage of using UIO is that we have to write a small kernel module or even none at all. However, other benefits also include [34]:

- only one small kernel module to maintain,
- the main part of the driver can be in user space,
- driver bugs will not crash the kernel,
- driver updates can take place without recompiling the kernel.

The UIO system offers a device driver called "uio_pdrv_genirq". This UIO driver implements a generic interrupt handler. By using it, we can avoid writing kernel code altogether. We must specify that the device is a UIO device and on which interrupt line it resides. All the rest can be handled in user space. One drawback of using this is that the interrupt line can not be shared.

2.5 Instruction set architecture

An instruction set architecture or ISA is everything visible to the programmer. It includes the set of instructions, the memory model etc. A realization of the ISA is called an implementation. One ISA can have several different implementations that vary in size, cost, and performance. However, all of these implementations can run the same software.

Instruction set architecture must be defined before starting the processor design. The problem here is that ISA design has direct implications on the design of the processor and its performance.

Generally, ISAs are classified as either a complex instruction set architecture (CISC) or a reduced instruction set architecture (RISC). The philosophy of the former ISA is to have many specialized instructions. Because we have so many different instructions, the compiler can choose the right one for the given task. However, the problem with CISC was that it made processors to complex.

The RISC philosophy is to reduce the number of different instructions and use those that are most commonly used by programs. Several RISC instructions thus replace more complex instructions of CISC. This fact makes RISC programs larger, which would mean that it would take longer for them to execute. However, this is not true because a RISC processor can be much faster (i.e., shorter clock period) and thus still outperform a CISC processor, which does not need to execute as many instructions.

2.5.1 RISC-V ISA

RISC-V is a free and open instruction set architecture (ISA) developed by the RISC-V Foundation [35]. The RISC-V started as an academic project at the University of California, Berkeley. However, since then, it has gained support from many big names in the industry like Google, Samsung, Nvidia, Alibaba, Siemens, Nokia.

"The guiding principles in defining the RISC-V architecture was to make the ISA suitable for nearly any computing device" [36]. RISC-V is thus versatile enough to be used for both low power embedded devices and high-performance computing. To enable such broad support, its ISA was split into the base ISA and optional ISA extensions. The base ISA is the minimum instruction set that each implementation must support. It is enough for many embedded processors, but still able to run modern software stacks. The base ISA can then be extended to enable support for computational workloads and multiprocessing [36].

The base ISAs of RISC-V is completed and ratified, but some extensions are still a work in progress. Nevertheless, many academic and even commercial processors for RISC-V have been made.

2.5.1.1 Base ISA

There are three different base ISAs: RV32I, RV32E, RV64I. RV64I and RV32I differ mainly in the size of the addressable memory space and the size and number of registers. RV32E is RV32I, with half as many general-purpose registers. RV32E is meant to be appropriate for low-end implementations. The RV32I is what we use in this thesis. Therefore, from now on we will describe only RV32I. It is a simple instruction set comprising of only 47 instructions. Still, this is enough to support high-level languages like C and even modern operating systems [36]. The instructions generally fall into four categories:

- system calls,
- arithmetic or computational instructions,
- control flow instructions,
- memory access instructions.

The base ISA does not contain support for hardware multiplication or division of integers. Neither does it offer any support for floating-point numbers.

The register file accommodates 31 general-purpose registers, a zero register, and a program counter register. All registers are 32-bit wide. Figure 2.15 shows the register file of the RV32I base instruction set.

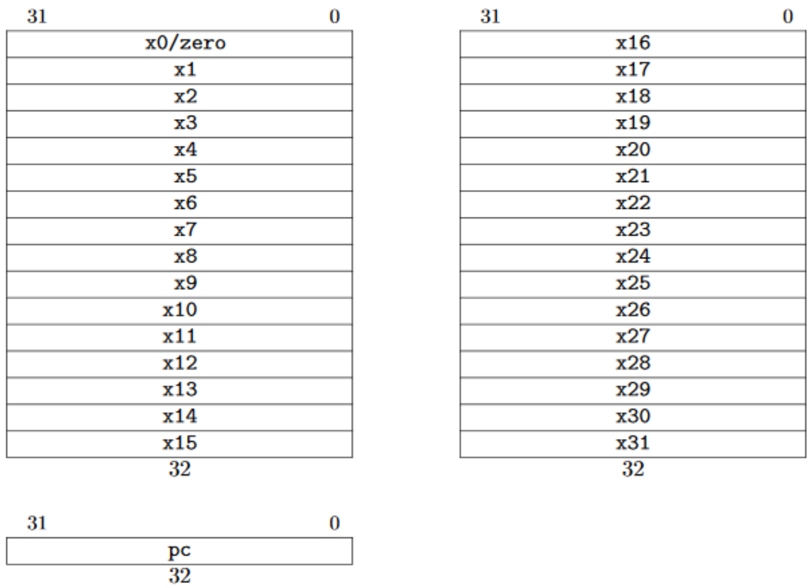


Figure 2.15: RV32I register file [36].

2.5.1.2 Standard ISA extensions

To extend the limited functionality of the base instruction set architecture, there exist various standard and non-standard extensions. RISC-V is a work in progress, so not all the extensions are finalized. Here are some extensions that are proposed:

- M for integer division and multiplication,
- A for atomic memory instructions,
- F single-precision floating-point arithmetic,

- D double-precision floating-point arithmetic,
- C support for compressed instructions,
- V the vector extension (not finalized),
- P the packed SIMD extension (not finalized).

The packed SIMD extension might be made obsolete by the V extension. An important note to make is that the floating-point extension introduces 32 extra registers. The floating-point unit can only use the floating-point register file. The decision to provide a separate register file does increase the die area, but it also enables higher performance.

As the title of the thesis suggests, it is also possible to add custom extensions. We have implemented such an extension. It is described in later chapters.

2.6 Loop unrolling

Loop unrolling is a compiler optimization on loops. The name is very self-explanatory, but it is even better illustrated by an example. Listing 2.2 shows an example of a loop that adds two vectors of size 16. Without using optimizations, a compiler might generate code that would contain the following assembler instructions to compute one iteration of the loop:

- two load instructions,
- one add instruction and another add instruction for the counter,
- a store instruction,
- and branch instruction.

What loop unrolling does is the following. Instead of having a loop, we just replace the loop by just add instructions (disregarding the load/store instructions). For Listing 2.2, this would be 16 add instructions. Of course, these increase the code size, but it also means we do not need branch instructions, and we do not need to handle the counter. Having fewer instructions to execute means, we could potentially compute the result quicker.

Partial loop unrolling is a compromise solution shown in Listing 2.3.

Listing 2.2: A normal for loop.

```
for (int i=0; i < 16 ; i++)
{
  a[i] = b[i] + c[i];
}
```

Listing 2.3: A partially unrolled for loop.

```
for (int i=0; i < 4; i=i+4)
{
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
```

Summary

In this chapter, we introduced the basics of neural networks. We discussed the mathematical model of the artificial neuron. We then extended this to layers of neural networks. We look in-depth into computing the fully connected and the convolutional layer. We also briefly discussed neural network architecture.

Next, we moved to FPGAs. We use them to emulate our microcontroller. We also elaborated AMBA. It is a bus specification that will allow us to connect various blocks of our system. The Linux operating system was also reviewed. It is used in a monitoring system for the microcontroller. This system is elaborated in Chapter 4. We also discussed the instruction set architectures. We finished with a quick look at the loop unrolling optimization. In the next chapter, we look at the microcontroller we use, its processor, and how to extend the ISA.

3.0 Hardware

In this chapter, we describe the microcontroller we are using to run neural networks. We take an in-depth look at the core of the microcontroller. We also show how we extended the microcontroller ISA.

3.1 PULPino

Instead of starting from scratch, we decided to use the open-source microcontroller called PULPino. The PULPino microcontroller is part of the Parallel Ultra Low Power (PULP) platform [37]. The PULP platform is a joint effort of the Integrated Systems Laboratory of ETH Zürich and Energy-efficient Embedded Systems group of the University of Bologna that aims to develop open-source hardware for ultra-low power applications. Figure 3.1 shows the block diagram of PULPino.

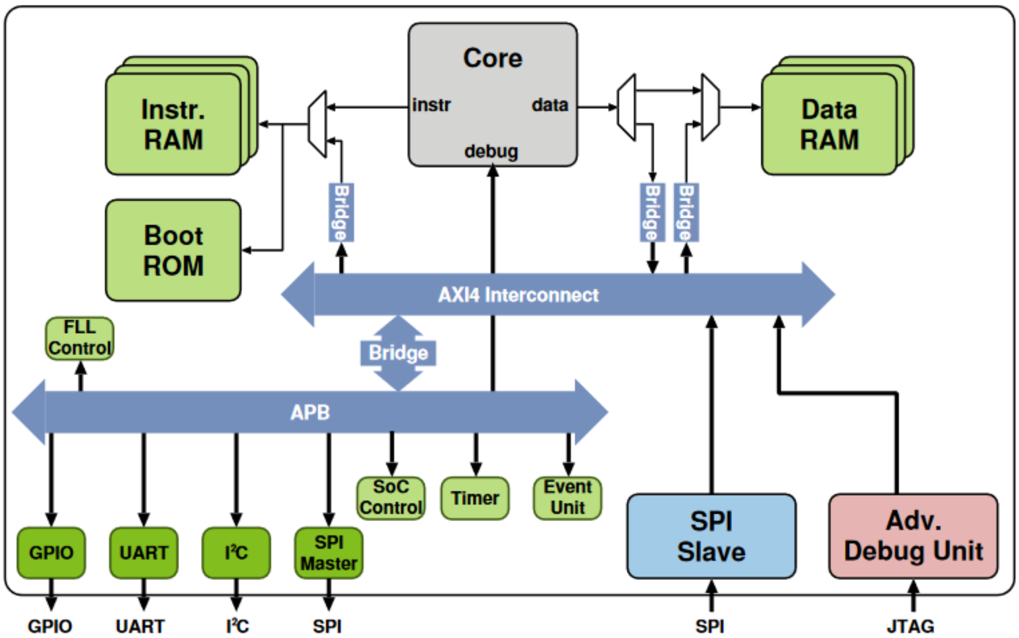


Figure 3.1: Block diagram of the PULPino microcontroller [38].

PULPino is mainly targeted at RTL simulation and application-specific integrated circuit (ASIC) synthesis. However, there is also an FPGA version. The FPGA version is not optimal in terms of performance, but it enables emulation of the device [38]. It comes with 32 KB of instruction memory and 32 KB of data memory. To program the memories, two interfaces are available: the SPI interface and the debug interface (i.e., JTAG). The design also includes a set of standard peripherals like UART, I2C, and SPI. PULPino comes with two different types of cores: RI5CY core and Zero-riscy. We chose the former core. The next section explains why.

3.2 The RI5CY core

RI5CY [1] is a small 32-bit 4-stage in-order RISC-V core which fully implements the RV32IMFC instruction set and also adds some other extensions. As explained in Chapter 2, this means the base RISC-V ISA and extensions for integer multiplication, floating-point numbers, and compressed instructions. Figure 3.2 shows the block diagram of the RI5CY core. RI5CY features an instruction cache that can hold a total of 128 instructions. It is the only cache in the system. RI5CY also has a dot product unit for integer operations. We chose to omit this in Figure 3.2 because it is not relevant to our design. The core's floating-point unit is optional and can be excluded from synthesis. The complexity of the core is typical of those found in microcontrollers. Other reasons for choosing this core are:

- It is open-source.
- It is low power.
- It supports floating-point numbers.
- It supports hardware loops.
- It has a relatively small footprint.

The RI5CY core has many optimizations to enable decent signal processing ability at a very tight power budget. The optimizations include [1]:

- hardware loop support,
- post-incrementing loads and stores,
- an optimized instruction fetch unit with an L0 buffer and support for hardware-loop handling,
- support for flexible fixed-point and saturated arithmetic operations as well as SIMD extensions.

The PULP architecture and the RI5CY core are designed for multicore operation. In [1],

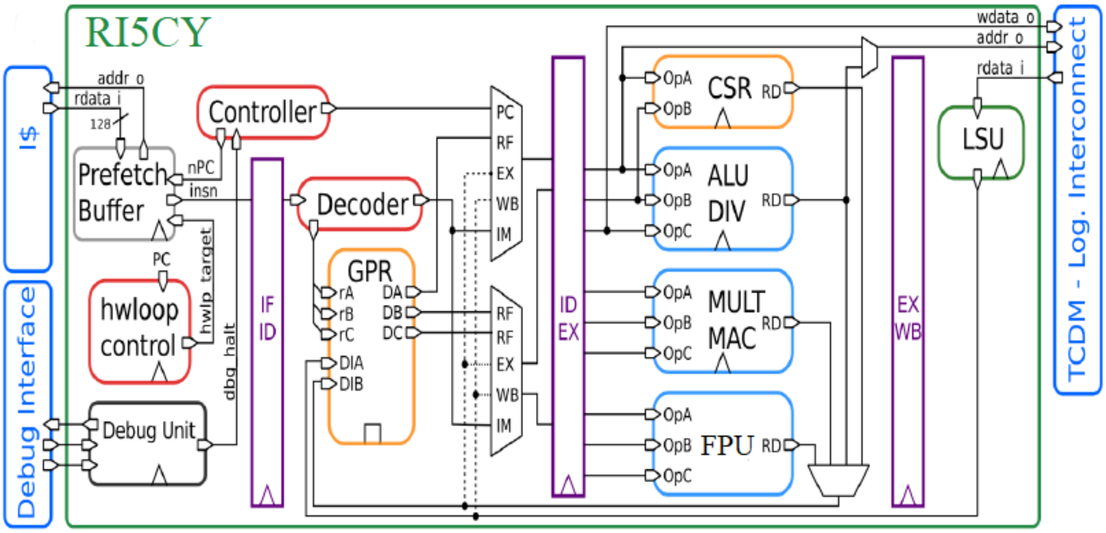


Figure 3.2: Block diagram of RI5CY core [39].

they discuss how a multicore configuration of such cores running at low frequency and a near-threshold voltage can run parallel algorithms with minimum power consumption. We will only use a single instance of the RI5CY core in our system, but the system can be extended to multicore.

3.2.1 Hardware loops

A significant extension supported by the RI5CY core is hardware loops. Hardware loops allow executing loops without the overhead of branches. Hardware loops involve zero stall cycles for jumping from the end to the start of the loop [39]. As our application contains many loops, we use this feature extensively. The core is also capable of nested hardware loops. However, the nesting is restricted to the depth of one.

Additionally, the instruction fetch unit of the RI5CY core is the hardware loop aware. It makes sure that the appropriate instructions are stored in the cache. This solution minimizes unnecessary instruction fetches to the main memory. Note that the hardware loop extension is not standard. The PULP platform also provides a modified GNU compiler that is capable of automatically using hardware loops.

A hardware loop is defined by its start address, its end address, and a counter that is decremented with each iteration of the body of the loop [39]. Listing 3.1 shows an assembly code that calculates the factorial of 5 and stores it in register x5. Remember that in RISC-V x0 is a special register hardwired to the constant 0.

We start by setting registers x5 and x6 to 1. We use x5 as the result register and x6 as a counter register. To set up the hardware loop, we use special "lp." instructions. With these instructions, we set the number of iterations, the start address, and the end address. Having set up the loop, the next two instructions *mul* and *addi* will get executed 5 times automatically. The *mul* instruction multiplies x5 with the value of the counter, and *addi* simply increments the counter. After five iterations of the loop, we compute the factorial of 5.

Note that the three "lp." instructions can also be replaced by a single instruction if the loop is small enough. The body of the loop must contain at least two instructions.

Listing 3.1: Assembly code for calculating the factorial of 5 using hardware loops.

```
addi x5, x0, 1
addi x6, x0, 1

lp.counti x1, 5
lp.starti  x1, start_HWLP
lp.endi    x1, end_HWLP

start_HWLP: mul x5, x5, x6
end_HWLP: addi x6, x6, 1
```

3.2.2 Performance counters

The RI5CY core comes with several special-purpose registers that enable monitoring of performance. They must be enabled in software. Table 3.1 lists some of the performance counters and a short description of them. Performance counters are placed inside the control and status register file and are accessible by using special software instructions or via the debug interface. The control and status register file can be accessed via the privileged instruction set. For example, the cssrw instructions perform an atomic read or write to a specified CSR register.

We will use these performance counters later to measure the performance of our code. This way, we will get actual performance data of the microcontroller.

Table 3.1: RI5CY performance counters [39].

Register	Name	Description
PCCR0	CYCLES	Counts the number of cycles the core was active (not sleeping).
PCCR1	INSTR	Counts the number of instructions executed.
PCCR2	LD_STALL	Number of load data hazards.
PCCR3	JR_STALL	Number of jump register data hazards.
PCCR4	IMISS	Cycles waiting for instruction fetches, i.e. number of instructions wasted due to non-ideal caching.
PCCR5	LD	Number of data memory loads executed. Misaligned accesses are counted twice.
PCCR6	ST	Number of data memory stores executed. Misaligned accesses are counted twice.
PCCR7	JUMP	Number of unconditional jumps (j, jal, jr, jalr).
PCCR8	BRANCH	Number of branches. Counts taken and not taken branches.
PCCR9	BTAKEN	Number of taken branches.
PCCR10	RVC	Number of compressed instructions executed.

3.3 The dot product unit

This section is a short description of the work of Mr. Karl Sturm, who modified the RTL code of the RI5CY core and added the instruction support to the GNU assembler.

As deep learning algorithms have many matrix operations, it makes sense to add special instructions for that. We decided to add an instruction that calculates a dot product on two vectors with up to four elements, where each element is a floating-point number. The output is a scalar single-precision floating-point number representing the dot product of the two vectors.

We did not implement any vector load instruction; instead, we use the standard load instruction for floating-point numbers. This decision has presented some problems as the compiler is not aware of the vector nature of our instructions and try's to rearrange them. This fact forced us to write assembly kernels by hand to use the unit.

The "×" in Figure 3.3 represents a floating-point multiplier, and the "+" sign represents a floating-point adder. The unit implemented two instructions which we added to the ISA:

- p.fdotp4.s - dot product of two 4-element vectors,
- p.fdotp2.s - dot product of two 2-element vectors.

When executing instruction p.fdotp2.s, the dot product unit automatically disconnects the terminals of a switch S, and similarly, it connects them when executing instruction p.fdotp4.s. For further information on the dot product unit, see [40], [41] and [41]. The work is later extended for basic linear algebra [42]–[47], and several applications [42], [43]. However, the dot product unit presented in [41], and [48] focuses on application that require very high-precision arithmetic. Also, the proposed dot product unit in the literature is deeply pipelined which is not necessary for our use-case [49].

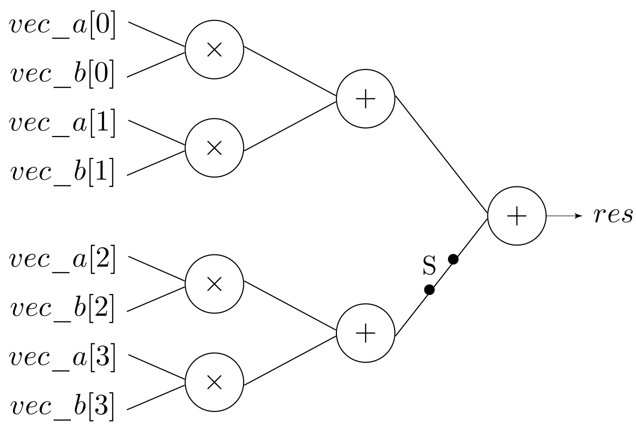


Figure 3.3: A schematic representation of the dot product unit.

The unit sits alongside the standard floating-point unit and uses the floating-point register file. The unit is not pipelined. Note that we did not follow the convention of the proposed vector extension for RISC-V. Figure 3.4 shows the pipeline of the modified RI5CY core. Further references to the modified RI5CY core reference the RI5CY core with the dot product unit.

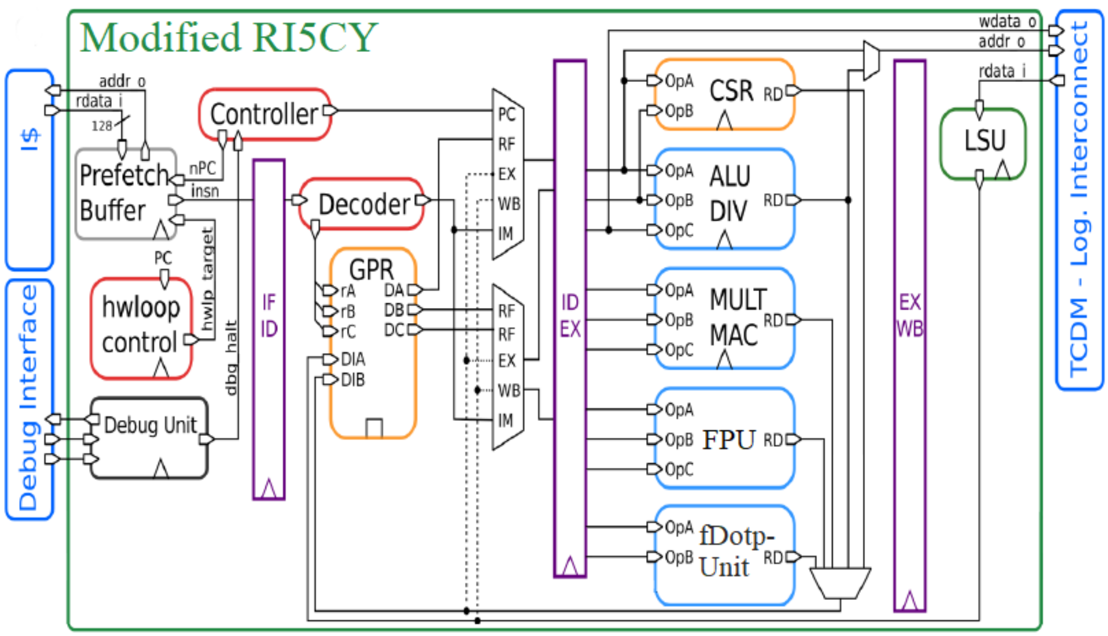


Figure 3.4: Block diagram of the modified RI5CY core [39].

Summary

In this chapter, we presented the hardware we are using to run neural networks. We described the PULPino microcontroller and the RI5CY core. After that, we presented the dot product unit we added to the RI5CY core. In the next chapter, we look at the system we designed to evaluate the performance of the microcontroller.

4.0 FPGA prototyping

To be able to test the performance of various deep learning algorithms running on the PULPino microcontroller, we develop a system for easy testing of such algorithms. We decided to use a Zynq-7000 system, which combines an ARM Cortex-A9 core and an FPGA on the same chip. With this setup, we can emulate any sensor using the data stored in DRAM. This data is then sent to the modified PULPino running on the FPGA. The purpose of the ARM Cortex-A9 core is to program, control, and monitor the PULPino. The subsequent section will shortly detail the Zynq-7000 family of SoCs. Next, we discuss the system design from the hardware perspective. Following that, we look at the software running on the ARM Cortex-A9. It includes the Linux operating system and a user space application. We finish with a short overview of all the software stacks in our system.

4.1 Hardware setup

4.1.1 The Zynq-7000 SoC family

The Zynq-7000 is a family of SoC products from Xilinx. They are a combination of an ARM processor and other hard intellectual property (IP) with an FPGA fabric. This product is a very flexible design that allows a general-purpose processor-based system to offload certain, specialized tasks to custom-designed hardware. The processor is either a single-core or a dual-core Cortex-A9. The Cortex-A9 cores are powerful application processing units that can run general-purpose operating systems like Linux. Figure 4.1 shows the block structure of the Zynq-7000 SoCs. The hard IP of the Zynq-7000 is referred to as the processing subsystem (PS), and the FPGA fabric is called the programmable logic (PL). In Figure 4.1, the bottom part of the picture is the PL, and the upper part is the PS. The PS contains not only the ARM cores but also caches, on-chip memory, external memory interfaces, and a rich set of peripherals. Some of these peripherals we connected to the modified RI5CY core in the PL part of the chip. In this configuration, they can emulate sensors that typically come with these types of interfaces. The PL part is a Xilinx 7 Series FPGA fabric. We are using the ZedBoard, which comes with Zynq-7000 SoC XC7Z020-CLG484-1. Here are some technical specifications of the PL side on the SoC above:

- an Artix-7 FPGA fabric,
- 85K programmable logic cells,
- 53,200 look-up tables (LUTs),
- 106,400 flip-flops,
- 4.9 Mb of block RAM[1],
- 220 DSP slices.

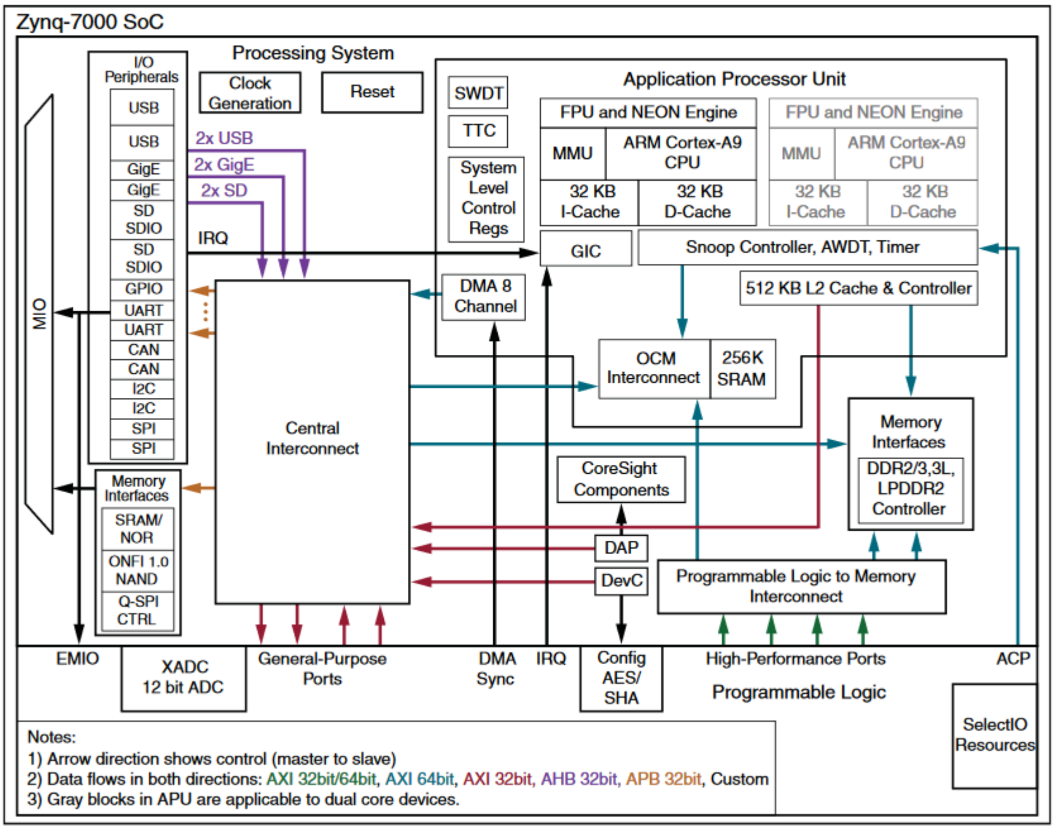


Figure 4.1: Block diagram of a Zynq-7000 SoC [50].

[1]That is bits, not bytes!

4.1.2 The ZedBoard

ZedBoard is a printed circuit board which features XC7Z020-CLG484-1, a member of the Zynq-7000 SoC family. Figure 4.2 shows the block diagram of the ZedBoard. It provides all the necessary interfaces to enable a wide range of applications:

- 512 MB DDR3 memory,
- 256 Mb Quad-SPI flash,
- SD card expansion,
- Ethernet,
- PMOD compatible extension ports,
- display capability via the HDMI port, VGA port or on the on-board OLED screen.

The ZedBoard can also be expanded. For this, it features PMOD connectors which can be used to attach sensors. That makes it well suited for rapid prototyping and proof-of-concept development. The ZedBoard is also affordable and is supported by a community website which features a forum and training materials. We use the ZedBoard to prototype our system. Furthermore, because the ZedBoard can be expanded, we can upgrade our design later to test it with actual data from sensors.

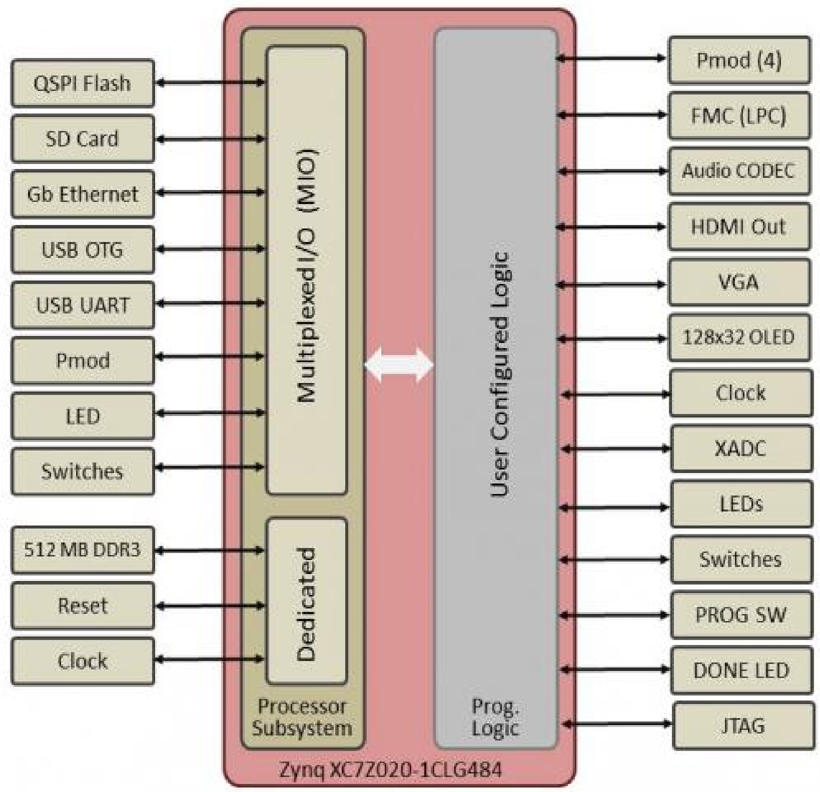


Figure 4.2: Block diagram of the ZedBoard [51].

Next, we look at the system design of our microcontroller testing system, starting with the hardware design and then the software design.

4.2 System design

Figure 4.3 shows the block diagram of the system. The diagram is split into two parts: the processing subsystem and the programmable logic part. On the PL side, we have the emulated microcontroller, and on the other side, we have the ARM core and the other hard intellectual property cores. In between, we have various interfaces that enable data transfer between both sides, including a UART interface, a QSPI interface, an AXI ACP bus, and an interrupt line.

We use the QSPI bus to do the initial programming of the PULPino memories. We do this because it was already set up in this way in the original PULPino project. The AXI ACP bus enables us to high-speed data transfers to the microcontroller. In this configuration, we can get some data from the DDR memory, process it, send back the results, and repeat. In the second stage, we could connect the PULPino directly to an actual sensor via the expansion ports on the ZedBoard.

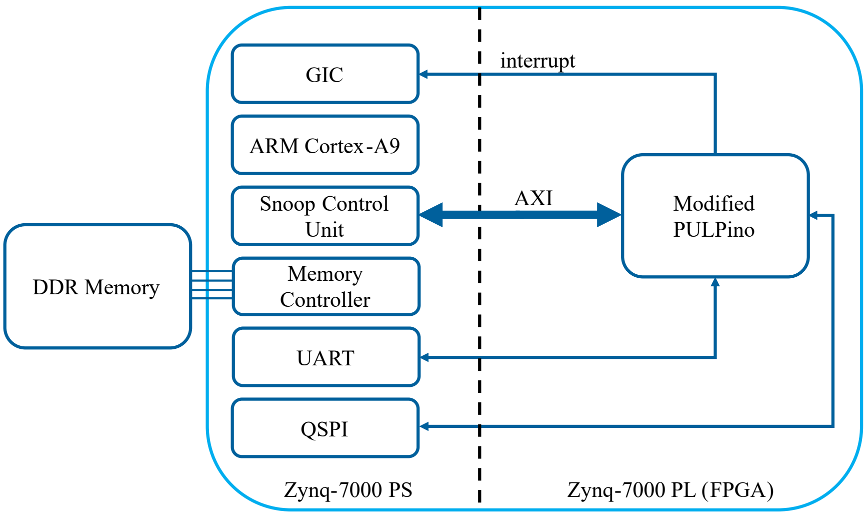


Figure 4.3: Block diagram of the system.

There are several AXI buses between the PS and PL side. We chose the AXI ACP bus to avoid problems with coherency. AXI ACP is the only one that ensures coherency with caches. This decision will simplify our software.

4.2.1 Porting to Vivado block design

The PULPino microcontroller is provided in RTL form and is targeted mainly towards ASIC synthesis. However, an FPGA version is also added, and it is targeted specifically for the ZedBoard. Upon trying to rebuild the design, we found that Vivado does not gracefully handle opening the designs from previous generation software. We use Vivado 2018.2, and the original FPGA project is built with Vivado 2015.1. We first reverted to old software, but later decided that it would be worth porting the project to a newer version of Vivado. We do this because we think it is easier to connect the PULPino and the ARM core with an AXI bus in block design.

The main problem we had with Vivado is that it does not offer any reasonable alternative for configuring the hard IP on the Zynq-7000 SoC except the block design. However, the block design does not work in between different versions of Vivado. Our solution was to recreate the entire project in a new block design. The result of this work is shown in Figure 4.4. To simplify the rebuilding process, we replaced some IP blocks from the original PULPino project. Namely, we replaced the clock generation block, the AXI interconnect, and the AXI to APB bridge.

4.2.2 Expanding the memory

The original PULPino comes with 32 KB of data memory and 32 KB of instruction memory. The amount of data memory here is a major problem for running neural networks. We discuss the size requirements for neural networks in Section 5.4.

Although PULPino does offer to select the memory size trough parameters, its implementation is incomplete for the FPGA synthesis flow. We fixed this issue by modifying PULPino RTL to use the Xilinx Parameterizable Macros for 7 Series Xilinx FPGA devices. Vivado recognizes these macros and inserts a predefined IP block in their place. This solution provided us with an easy way of changing the memory size by merely changing parameters in the top-level SystemVerilog file.

4.3 Processing subsystem software design

We decided to run Linux on the ARM side. This way, we get lots of features like interrupt handling, serial messaging, and time measurement virtually for free. The negative side of using an operating system is that it enables the memory management unit. Such an approach prevents unauthorized access to the DDR memory from the PL side. The next subsection will

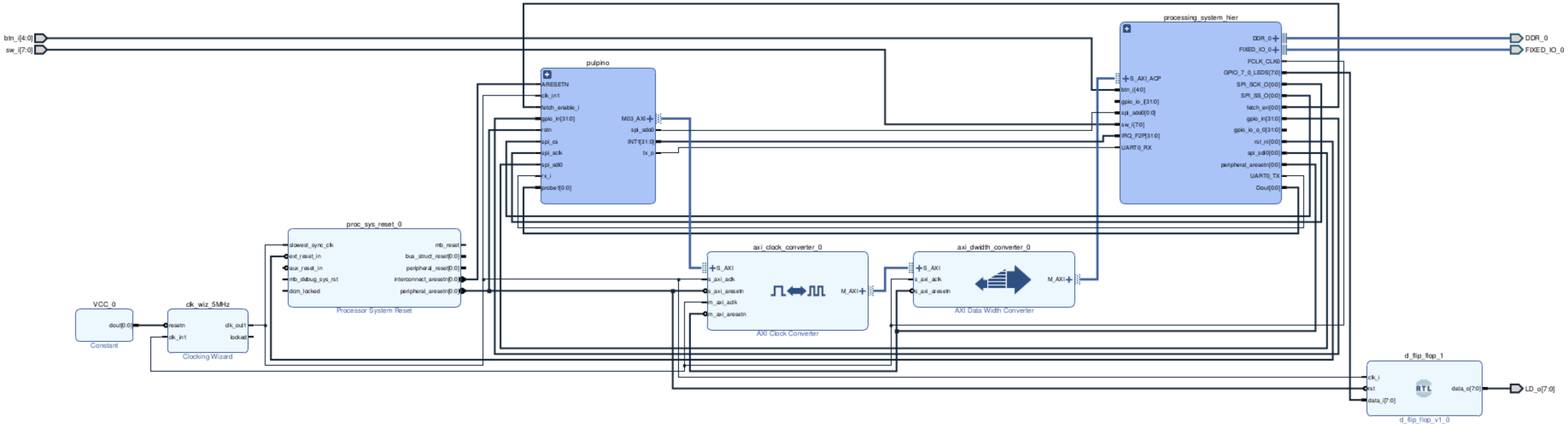


Figure 4.4: Snapshot of our design in Vivado block design.

show how to tell Linux to authorize access to that memory and to recognize an interrupt from the PL side.

4.3.1 Customizing Linux

We generate our Linux image using the PetaLinux tools. It is a toolset that simplifies customizing, building, and deploying Linux solutions on Xilinx processing systems. PetaLinux tools are provided free of charge by Xilinx.

The first thing we had to do was to customize Linux for our board. The easiest way to do this was to download the board support package from the Xilinx website [52]. With this input, the PetaLinux tools create a new Linux project that is aware of all the hardware available on the ZedBoard.

After that, we must customize it further. First, we must enable the UIO drivers. We decided to link in statically the UIO drivers[2]. Next, we had to modify the device tree file. In PetaLinux you can add your modification to the device tree by modifying the device tree file ”system-user.dtsi”. This file gets appended to the end of the auto-generated device tree files.

Listing 4.1 shows a part of the device tree we modified. The first entry called ”reserved-memory” in Listing 4.1 reserves a 128 MB memory region in DRAM at address $0x10000000$. We use this to test large neural networks that need large amounts of data. The second entry, called "res_mem_driver" does two things. First, it registers a device as a UIO device. Second, it specifies which interrupt line this device is using. It will tell the Linux kernel, which interrupts to pass to the user space application. The keyword "compatible" is set to "generic-uio". It tells the kernel which driver to use — the UIO driver.

After modifying the Linux, we generated an image and booted the Linux trough an SD card. By following this procedure, the UIO device must be listed in the /dev folder. Linux uses this folder to list all detected devices. Our device, for example, would create a file called uio0.

[2]We could have also used something called kernel modules and load the module in at runtime.

Listing 4.1: Device tree modification.

```
reserved-memory {
                device-type = "reserved_memory";
                #address-cells = <1>;
                #size-cells = <1>;
                ranges;
                pl_reserved:buffer@0{
                        no-map;
                        reg = <0x10000000 0x08000000>;
                };
        };

res_mem_driver: buffer_driver@0{
                #address-cells = <1>;
                #size-cells    = <1>;
                compatible = "generic-uio";
                memory-region = <&pl_reserved>;
                reg = <0x10000000 0x08000000>;
                #interrupt-cells = <2>;
                interrupt-controller ;
                interrupt-parent = <&intc>;
                interrupts = <0 29 1>;
        };
};
```

4.3.2 The user space application

On the ARM core, we are running Linux, and also on top of that, we are running a user space application. The job of the user space application is to control the modified PULPino, provide it with data, and gather performance statistics. Modified PULPino is the PULPino with the modified RI5CY core that we ported to Vivado block design.

Algorithm 4.1 details the function of the user space application. The application has two inputs. The first one is the program image. It contains data and instructions that get copied to PULPino memories. The second input to the application is the data that is the input to the neural network being run on the modified RI5CY core. Data can be image data, sound

Algorithm 4.1: The users space application.

```
input : PULPino memory image memImg
input : sensor data pics of size pic × n

map_memory();
fdev = map_pulpino();
reset_pulpino();
program_pulpino_memory(memImg);
for i = 0 to n do
    // Copies data to the reserved − memory region which the RI5CY core can access.
    copy_data(reserved_memory, pics[n]);
    start_pulpino();
    wait_on_interrupt(fdev);
    read_results();
    reenable_interrupts(fdev);
    reset_ri5cy(); // This resets the core but not the memory.
print_performance_results();
close_device(fdev);
unmap_memory();
```

data, sensor data or generally any data that we input to the neural network.

The application starts by mapping the memory region we reserved in the device tree. The mapping is performed by a Linux system call ”mmap”. Next, we map the PULPino as a UIO device. It returns a handle to an application that we can use then. What follows is the handling of PULPino. We first reset it and program the memories using the input memory image. We use the SPI interface.

Finally, we start running the microcontroller. The microcontroller acts as a bus master and reads a chunk of data from memory. A chunk of data is, for example, a single image. The microcontroller then enables the performance counters and starts executing its program. When the microcontroller finishes it first stops the performance counters, it sends back performance data to the DDR memory and sets an interrupt to signal to the ARM core that it has finished.

After the ARM core receives an interrupt, it reads and stores the performance data of the run. It then resets the microcontroller and repeats the process as many times as we wish (or for as long as we have simulation data). The user space application running on ARM must also reenable the interrupts. In the UIO framework, this is done simply by writing to the device file ”/dev/uio0”.

4.3.3 Software stacks

Because we have several pieces of software, we chose to add this section that provides a short overview of the various software. Figure 4.5 shows the different pieces of software we are running on the system. The programmable logic is the Zynq-7000 FPGA subsystem. On the processing subsystem side, we have Linux and the user space application. The job of those is to control and monitor the programmable logic side. On the programmable logic side, we have the neural network which uses the deep learning library. The neural network and the deep learning library are explained in Chapter 5.

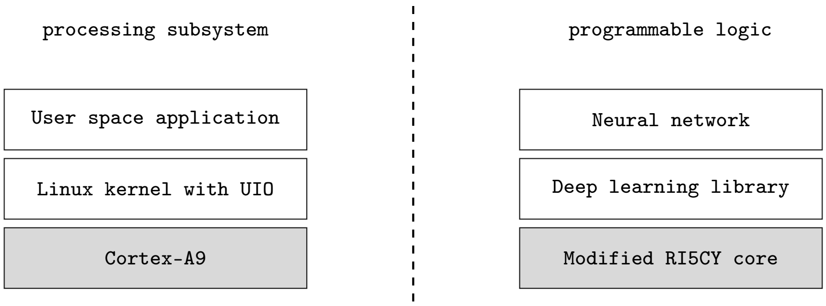


Figure 4.5: Overview of the software stacks.

Summary

In this chapter, we presented the system we designed for testing the neural networks running on microcontrollers. We first discuss the Zynq-7000 family of SoC and the ZedBoard. We then progressed to the system design, first discussing the hardware aspects of the design and then moving on to the software design. We are using the Linux operating system to program, control, and monitor the microcontroller. In the next chapter, we look at the deep learning library and the test neural network we developed to run on the microcontroller.

5.0 Neural network

In this chapter, we present the neural network we developed as a demonstration neural network. Following that, we describe the deep learning library and analyze the algorithms we implemented to utilize the dot product unit fully. We finish by listing some embedded-friendly neural networks.

5.1 Optical character recognition

As a demonstration, we developed a simple optical character recognition (OCR) neural network. Because of the small amount of memory available on the FPGA, the neural network has only five layers:

- convolutional layer + ReLU,
- maximum pooling layer,
- first fully connected layer + ReLU,
- second fully connected layer,
- softmax layer.

The primary purpose of this network is to have a relatively small network to test our system. The full architecture can be seen in Figure 5.1. Even though the network is relatively small, it still resembles common CNN architectures. It is just scaled down.

The network was trained in the TensorFlow framework using the MNIST dataset [53]. MNIST[1] contains more than 60,000 instances of 28×28 mono-colored images of handwritten digit characters. Figure 5.2 shows an enlarged picture from the MNIST dataset.

We achieved a satisfactory 95% accuracy. Much better results can be obtained for the MNIST dataset, but the object of this thesis is not a neural network architecture design. The output of the network is a vector of ten floating-point numbers which represent the probability that the corresponding index of the vector is the digit on the image. In total, the network contains 9,956

[1]MNIST is offered under a Creative Commons Attribution-Share Alike 3.0 license.

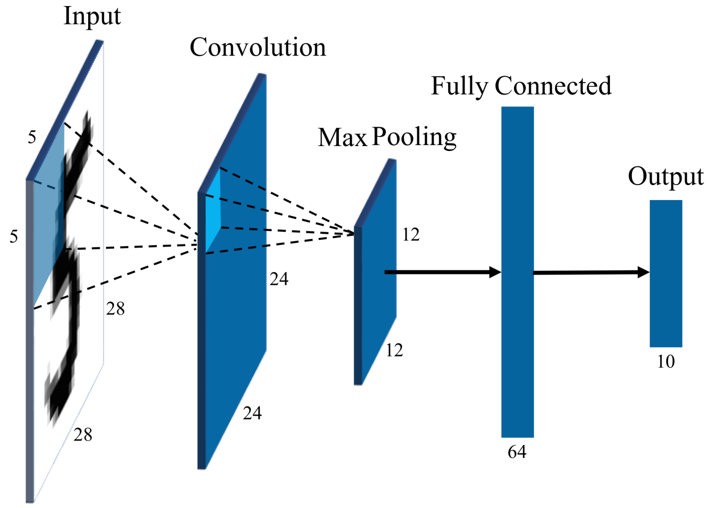


Figure 5.1: Architecture of the demonstration software.

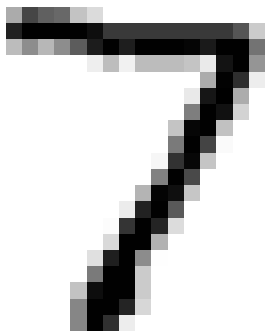


Figure 5.2: An example picture from the MNIST dataset [53].

parameters, which consume roughly 39 KB of memory space if we use the single-precision floating-point data type. To compute one pass of the network, we must make 24,256 multiply and accumulate (MAC) operations.

5.2 Porting the neural network

Getting the neural network on the RI5CY core was a challenge. TensorFlow 2.0 does support porting the models to arbitrary architectures by exporting generic C++ code using an experimental port called TensorFlow Lite for Microcontrollers. However, at the time, we were not aware of this and decided to do it manually. Another reason for doing it manually is that we maintain complete control over the execution of the software and thus can better tweak its performance.

After training the model in TensorFlow, we exported the weights as a Hierarchical Data Format (HDF). This format is used to store and organize large amounts of data. We then imported the HDF file into a new Python script and used this script to generate a set of C header files that contained arrays of weights. We applied the same procedure to the input data. TensorFlow has custom functions to access various standard datasets. They made it easier just to obtain our data using TensorFlow.

After having the weights stored conveniently in header files, we needed to write the software to use them. We wrote a library that can perform a limited set of deep learning algorithms. This library is detailed in the following section.

5.3 Deep learning library

We first implemented the deep learning functions using naive algorithms and without using the dot product unit. We name this version of the library the reference version. Following that, we wrote the optimized code that uses the dot product unit. To make efficient use of the added dot product unit, we used inline assembly to have better control over the program. We name this version of the library the optimized assembly version. Because of the repeating nature of the computation, we also used the hardware loop extension, which minimizes the overhead of loops. Table 5.1 lists the functions we implemented in our library.

Table 5.1: Functions in our deep learning library.

Function	Description
dlDensen	Performs matrix-vector multiplication between an arbitrary matrix and a vector and adds the bias.
dlDensenwReLU	Performs matrix-vector multiplication between an arbitrary matrix and vector, adds the bias, and performs the ReLU function over the result vector.
dlConv2n	Performs neural network style convolution over an arbitrary input image and a filter.
dlConv2nwReLU	Performs neural network style convolution over an arbitrary input image and a filter. It also performs the ReLU function over the output.
dlMaxPool	Computes the maximum pooling function.
dlReLU	Computes the ReLU function on an arbitrary sized input vector.
dlSoftmax	Computes the softmax function for a given input vector.

Most of the computing time is spent in computing either convolution or matrix-vector multiplication (computing fully connected layers requires computing a matrix-vector product). For this reason, we only wrote inline assembly code that uses the dot product for those functions (dlDensen, dlDensenwReLU, dlConv2n, dlConv2nwReLU). The rest of the functions do not use the added dot product extensions. It is widespread to have a combination of dlConv2n and dlReLU. It is why we decided to add a function that can be optimized to run the two previously stated functions. The implementations of the naive versions of the functions can be concluded by looking at Section 2.1. The optimized algorithms are discussed in the subsections that follow.

5.3.1 Matrix-vector multiplication with bias addition

Algorithm 5.1 and Figure 5.3 illustrate a matrix-vector multiplication. We name the matrix and vector, on which we are performing the matrix-vector multiplication, the input matrix M and the input vector $\vec{v}$ in order to separate them from the vectors of the dot product unit. Additionally, we signify the bias vector as $\vec{b}$.

The approach we used was to load in a chunk of the input vector $\vec{v}$ and calculate each product for that chunk. The size of the chunk is determined by the dot product unit input size; in our case, it is 4. One problem with this approach is that the width of the matrix must be a multiple of four. This problem can be circumvented by zero-padding the matrix and vector.

Equation (5.1) elaborates our approach for a 4x8 input matrix M, an 8x1 input vector $\vec{v}$ and a 4x1 bias term $\vec{b}$. Subscripted vectors $\vec{m_n}$ and $\vec{v_n}$ represent the chunks of M and $\vec{v}$. Each chunk is a vector with 4 elements. Consider that the matrix and vector elements from Equation (5.1) are stored in memory, as shown in Figure 5.3. As seen in Algorithm 5.1, we first load in $\vec{v_0}$, following that we load in vectors $\vec{m_0}$, $\vec{m_1}$, $\vec{m_2}$, $\vec{m_3}$. We then move on to loading in $\vec{v_1}$ and vectors $\vec{m_4}$, $\vec{m_5}$, $\vec{m_6}$, $\vec{m_7}$. We compute the dot products in the following order: $\vec{m_0}^T \cdot \vec{v_0}$, $\vec{m_1}^T \cdot \vec{v_0}$, $\vec{m_2}^T \cdot \vec{v_0}$, $\vec{m_3}^T \cdot \vec{v_0}$, $\vec{m_4}^T \cdot \vec{v_1}$, $\vec{m_5}^T \cdot \vec{v_1}$, $\vec{m_6}^T \cdot \vec{v_1}$, $\vec{m_7}^T \cdot \vec{v_1}$. This way we only have to load in vectors $\vec{v_0}$ and $\vec{v_1}$ once and simultaneously, we have a good spatial locality of memory accesses.

$$M \cdot \vec{v} + \vec{b} = \begin{bmatrix} \vec{m_0}^T & \vec{m_4}^T \\ \vec{m_1}^T & \vec{m_5}^T \\ \vec{m_2}^T & \vec{m_6}^T \\ \vec{m_3}^T & \vec{m_7}^T \end{bmatrix} \cdot \begin{bmatrix} \vec{v_0} \\ \vec{v_1} \end{bmatrix} + \vec{b} = \begin{bmatrix} \vec{m_0}^T \cdot \vec{v_0} + \vec{m_4}^T \cdot \vec{v_1} \\ \vec{m_1}^T \cdot \vec{v_0} + \vec{m_5}^T \cdot \vec{v_1} \\ \vec{m_2}^T \cdot \vec{v_0} + \vec{m_6}^T \cdot \vec{v_1} \\ \vec{m_3}^T \cdot \vec{v_0} + \vec{m_7}^T \cdot \vec{v_1} \end{bmatrix} + \vec{b} \tag{5.1}$$

Algorithm 5.1 shows the approach in pseudo-code. The functions $load\_vecA()$ and $load\_vecB()$ take as input the pointer to the memory location from which to load 4 consecutive scalars.

Function load_vecA() is called first — it loads a chunk of the input vector. Next, the inner for loop traverses through a part of the input matrix. With load_vecB(), we load in chunks of the input matrix. We call dot_prod_vecA_vecB(), following each load of a chunk of the input matrix.

The approach we use offers a natural way to implement a multicore algorithm for matrix-vector multiplication.

Algorithm 5.1: Computing a fully connected layer with the ReLU non-linearity applied (dlDensenwReLU).

---

input : pointer to matrix M of size $m \times k$
input : pointer to vector v of size k
input : pointer to bias b of size m
output : pointer to result vector r of size m

for $i = 0$ **to** $(k/4) - 1$ **do**
    $load\_vecA(v + i * 4)$;
    for $j = 0$ **to** $m - 1$ **do**
        $load\_vecB(M + i * m * 4 + j * 4)$;
        $r[j] = r[j] + dot\_prod\_vecA\_vecB()$;

for $i = 0$ **to** $m - 1$ **do**
    $r[i] = r[i] + b[i]$;
    if $r[i] < 0$ **then**
        $r[i] = 0$;

	0x0009 FFF0				
M→	0x000A 0000	$m_0[0]$	$m_0[1]$	$m_0[2]$	$m_0[3]$
	0x000A 0010	$m_1[0]$	$m_1[1]$	$m_1[2]$	$m_1[3]$
	0x000A 0020	$m_2[0]$	$m_2[1]$	$m_2[2]$	$m_2[3]$
	0x000A 0030	$m_3[0]$	$m_3[1]$	$m_3[2]$	$m_3[3]$
	0x000A 0040	$m_4[0]$	$m_4[1]$	$m_4[2]$	$m_4[3]$
	0x000A 0050	$m_5[0]$	$m_5[1]$	$m_5[2]$	$m_5[3]$
	0x000A 0060	$m_6[0]$	$m_6[1]$	$m_6[2]$	$m_6[3]$
	0x000A 0070	$m_7[0]$	$m_7[1]$	$m_7[2]$	$m_7[3]$
	0x000A 0080				

	0x000A FFFC				
v→	0x000B 0000	$v_0[0]$	$v_0[1]$	$v_0[2]$	$v_0[3]$
	0x000B 0010	$v_1[0]$	$v_1[1]$	$v_1[2]$	$v_1[3]$
	0x000B 0020				

Figure 5.3: Memory layout for the matrix-vector multiplication algorithm.

5.3.2 Convolution operation

The strategy we use for speeding up convolution is always to keep the filter in the register file. We first save the contents of the register file to the stack. Such an approach enables us to use the entire register file without breaking the calling convention used by the compiler. Next, we

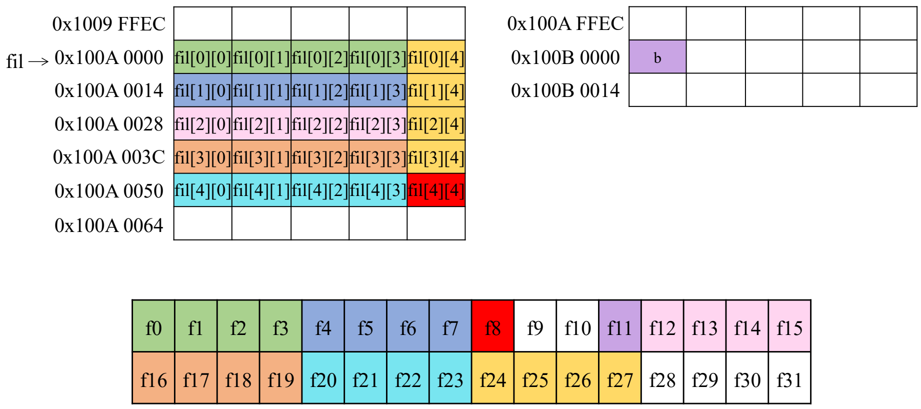


Figure 5.4: Allocation of the 5 × 5 convolution filter and bias term represented with color. The top part of the image shows the filter and bias term in main memory. The bottom part represents the floating point register file.

load the entire 5 by 5 filter and the bias term into the floating-point register file. RISC-V has 32 floating-point registers, so we have enough room in the register file to store a 5 by 5 filter, a chunk[2] of the image, and still have 2 registers left. Figure 5.4 shows how we loaded a 5 by 5 filter and a bias term into the floating-point register file. Registers f28-f31 were used to store chunks of the image and registers f9 and f10 to store the result.

Having the filter and bias term in the register file, we load in one chunk of the image at a time and compute the dot product with the appropriate vector of weights already stored in the register file. After traversing the entire image, we restore the register state. We make use of hardware loops to implement looping behavior.

Algorithm 5.2 details our algorithm. The input to the algorithms are:

- image - Is the image on which we are performing the algorithm — can be any size.
- filter - Is the filter we are using to perform the convolution. We are showing a 5 by 5 example, which is the maximum size of the filter for the 32 registers we have available.
- bias - A single floating-point number.
- stride - Is the size of the step we make while moving the filter across the image. The stride can be different in the horizontal and vertical axes.

The output of the algorithm is, as explained in Subsubsection 2.1.2.2, a matrix of floating-point numbers.

The functions load_vec_fn0_n1 load a total of (n1-n0) consecutive floating-point numbers from the location given in the argument to the registers n0 to n1. The function load_fn0 loads

[2] We again use the word chunk to refer to a part of an image in vector form. In other words, 4 floating-point numbers.

Algorithm 5.2: Computing the convolutional layer with the ReLU non-linearity (dlConv2nwReLU).

```
input  : pointer to image img of size img_size × img_size
input  : pointer to a filter fil of size 5 × 5
input  : bias b of size 1
input  : stride stride
output : result res of size out_size × out_size

saveRegisterState();
loadFilterAndBiasToRegFile();
out_size = ((img_size - fil_size)/stride) + 1;
// x and y get incremented by stride after each
// iteration
for x = 0 to out_size do
    for y = 0 to out_size do
        img_slice ← img[x][y];
        load_vec_f28_31(img_slice[0][0]);
        res[x][y] = dot_prod_f0_f28();
        load_vec_f28_31(img_slice[1][0]);
        res[x][y] = res[x][y] + dot_prod_f4_f28();
        load_vec_f28_31(img_slice[2][0]);
        res[x][y] = res[x][y] + dot_prod_f12_f28();
        load_vec_f28_31(img_slice[3][0]);
        res[x][y] = res[x][y] + dot_prod_f16_f28();
        load_vec_f28_31(img_slice[4][0]);
        res[x][y] = res[x][y] + dot_prod_f20_f28();
        load_f28(img_slice[0][4]);
        load_f29(img_slice[1][4]);
        load_f30(img_slice[2][4]);
        load_f31(img_slice[3][4]);
        res[x][y] = res[x][y] + dot_prod_f24_f28();
        load_f28(img_slice[4][4]);
        res[x][y] = res[x][y] + dot_prod_f8_f28();
        res[x][y] = res[x][y] + f11;
        if res[x][y] < 0 then
            res[x][y] = 0;
restoreRegisterState();
```

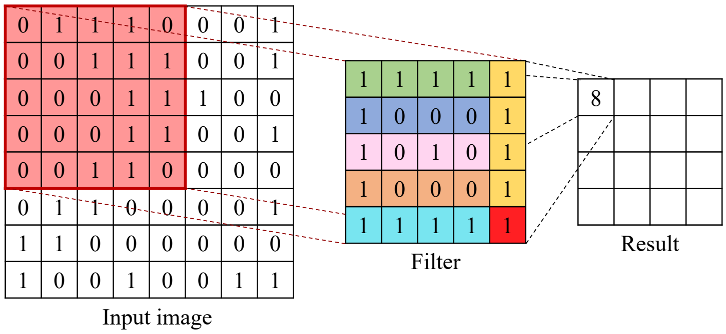


Figure 5.5: A visualization of the convolution operation using the dot product unit.

a single floating-point number to register n0. The functions dot_product_fn0_n1 compute the dot product between two chunks in the register file. The first chunk starts at n0 and ends at n0+3. The second chunk starts at n1 and ends at n1+3.

Figure 5.5 shows Algorithm 5.2 in action. The leftmost side matrix in the figure represents the input image, the middle matrix is the filter, and the rightmost matrix is the result matrix. The image shows the algorithm in the first iteration of line 22 in Algorithm 5.2. The darker red and blue colors represent the currently active vectors (i.e., the vectors being multiplied).

5.4 Embedded friendly neural networks

In this section, we present some data on the size of the networks. As a reference, we also show some other networks which are considered embedded friendly. By this, we mean that their memory footprint and computational complexity are within what many microcontrollers can handle. Table 5.2 lists such practical examples of embedded friendly neural networks.

Summary

In this chapter, we presented our demonstration neural network. It is a small network with a total of five layers. Following that, we illustrated the challenges of porting the neural network to our system. Next, we start describing our deep learning library. We made two versions of the library. One is the reference code, and the other is an optimized inline assembly code that takes advantage of the dot product unit. The next chapter shows the results we achieved by running this neural network on our system.

Table 5.2: Examples of embedded friendly neural networks.

Example network	Memory footprint	Data type	MAC operations	Description
MNIST-NN	39 KB	Float32	24256	Our example network for recognizing handwritten digits.
FastGRNN [54]	1 KB	8-bit integer	/	Small network capable of detecting a single activation keyword like "Hey Cortana".
Micro Speech [55]	22 KB	8-bit integer	/	Small example network that comes with TensorFlow Lite. It can detect keywords in audio data.
EdgeSpeechNet-C [56]	121.2 KB	Float32	83.5M	Speech recognition network with 96.2% accuracy on the limited-vocabulary Google Speech Commands dataset.
EdgeSpeechNet-D [56]	321.2 KB	Float32	24.5M	Similar to EdgeSpeechNet-C but with 95.8% accuracy.
Micro Vision [57]	250 KB	8-bit integer	/	Small example network that comes with TensorFlow Lite. It can detect people in camera images.
SqueezeNet [58]	0.47 MB	6-bit integer	/	Very compressed network with 57.5% top-1 accuracy and 80.3% top-5 accuracy on ImageNet.
Slim-CNN [59]	7.5 MB	Float32	/	Face recognition network with 91.24% accuracy on the CelebA dataset
MobileNetV2 [60]	13,6 MB	Float32	300M	Object detection network with 72.0% top-1 accuracy on ImageNet.

6.0 Results and analysis

This chapter presents the results of running neural networks on the original and modified PULPino microcontrollers. We start by discussing the ASIC synthesis results for the unmodified and modified PULPino. We add the synthesis result so that the reader can get an idea about the cost of adding such extensions. Next, we discuss the methodology we used to evaluate the neural network running on the unmodified and modified PULPino. We try out different optimizations for the neural network. We also dissertate the code size of the deep learning library. The code size is significant for microcontrollers, where memory is a scarce resource. Finally, we list and analyze the results for the fully connected layer, convolutional layer, and the entire neural network.

6.1 Synthesis results

Before we get into numbers, let us first discuss what we had to pay in the area and power to expand the RI5CY core with the dot product unit. These numbers were provided by Mr. Karl Sturm, who ran synthesis using the following setup:

- Synopsys Design Compiler O-2018.06-SP5,
- 90 nm generic core cell library from the United Microelectronics Corporation.

Table 6.1 shows the results of the synthesis. We see that the area increased by 72%. This area increase could have been smaller if we had reused one of the existing floating-point multipliers and adders[1]. The dynamic power results are only approximate. The dynamic power is reported by the Design Compiler using random equal probability inputs.

The static power consumption has more than doubled, but the dynamic power increased only slightly. Even though the dynamic power is three orders of magnitude larger than the static power, its rise may still be problematic in embedded devices that stay most of the time in

[1] The PULPino already has a floating-point unit. We added 4 extra multipliers and 3 adders. We could have instead reused the existing FPU for the one multiplier and one adder.

standby mode. This issue can be addressed by merely turning off the dot product unit while in standby mode.

Table 6.1: Synthesis results.

	Area[1] [kGE]	Area [μm^2]	Static power [μW]	Dynamic power [mW]
Original PULPino	77274.50	242332.83	307.13	147.78
Modified PULPino	133159.50	417588.19	640.37	148.47

[1] kGE signifies kilo-gate equivalent

6.2 Methodology

To gather data about the performance, we used the performance counters embedded in the RI5CY core. We compare and analyze our implementation in the following metrics:

- Cycles — how many clock cycles to compute the neural network,
- Instr — number of instructions executed,
- Loads — number of loads to the data memory,
- Stores — number of stores to the data memory,
- Jumps — number of unconditional jump instructions executed,
- Branch — number of branches,
- Taken branches — number of taken branches as a ratio to all branches.

We compare five different implementations. They are listed in Table 6.2. All five of them compute the same result.

The implementations FpHw, Fp, and FpHwU, run on the original PULPino. FpDotHw and FpDotHwU run on the modified PULPino. We chose the Fp implementation as the baseline. We aim to study how much the hardware loops, loop unrolling, and the dot product unit aid in speeding up the computation. The dot product unit is not used in all functions of our library. It is only used for Matrix-vector multiplication and convolution. However, these functions represent the vast majority of the computing effort. For details, see Section 6.3.

To compile our neural network, we used the modified GNU toolchain 5.2.0 (riscv32-unknown-elf-). The modified toolchain is provided as part of the PULP platform. It supports custom extensions, such as hardware loops. It automatically applies hardware loops when compiling

Table 6.2: Different implementations of the program.

Implementation	GCC compiler flags	Description
Fp	-O3 -m32 -g -mnohwloop	Reference library version with all optimizations except hardware loops.
FpHw	-O3 -m32 -g	Reference library version with all optimizations turned on.
FpHwU	-O3 -m32 -g -funroll-all-loops	Reference library version with all optimizations and loop unrolling.
FpDotHw	-O3 -m32 -g	Optimized assembly library version that uses the dot product unit and all optimizations turned on for functions that are not written in inline assembly.
FpDotHwU	-O3 -m32 -g -funroll-all-loops	Optimized assembly library version that uses the dot product unit and all optimizations and loop unrolling turned on for functions that are not written in inline assembly.

the neural network with optimizations enabled. Hardware loops can be explicitly disabled using the compiler flag -mnohwloop. The neural networks were compiled with the compiler flags that are listed in Table 6.2. Matrix-vector multiplication and convolution functions of the dot product unit are written in inline assembly. They always use hardware loops as the compiler does not optimize that code in any way. We did not use compressed instructions. As code size is a significant concern in embedded systems, we will look at the code size of different versions of our deep learning library in the next section.

6.3 Code size comparison

In this section, we present the code size comparison. We compare the size of the code for the functions listed in Table 5.1. The results are shown in Table 6.3. The first three implementations (Fp, FpHw, and FpHwU) use the reference version of the library, while the FpDotHw and FpDotHwU use the assembly optimized version of the library. Note that not all functions of the library were implemented in inline assembly. The cells highlighted in blue in Table 6.3 show the functions that were written in inline assembly. Compiler optimizations do not affect these functions.

In Table 6.3, where there are the same sizes in two cells of the same row, the assembly functions were identical at the assembly level.

The code size is best in the implementation where we do not use hardware loops or loop

Table 6.3: Code size of functions in our deep learning library. All results are in bytes.

Functions / Implementations	Fp	FpHw	FpHwU	FpDotHw	FpDotHwU
dlDensen	148	192	352	352	352
dlDensenwReLU	168	208	696	360	360
dlConv2n	196	200	576	912	912
dlConv2nwReLU	196	200	576	200	200
dlMaxPool	172	180	568	180	568
dlReLU	56	68	472	68	472
dlSoftmax	248	256	1188	256	1188

unrolling. As it is predictable, the code with loop unrolling (including our optimized inline assembly code) is the largest. Nevertheless, the code size is still quite small and in the realm of what most microcontrollers can handle. Figure 6.1 shows the total code size of various versions of the deep learning library.

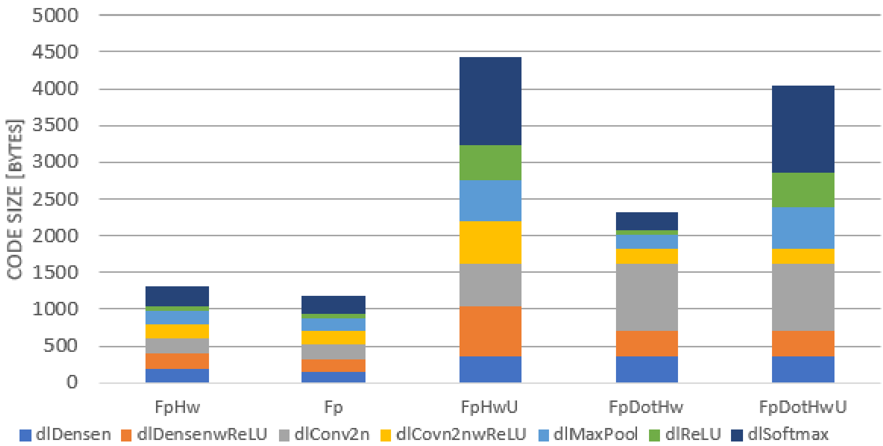
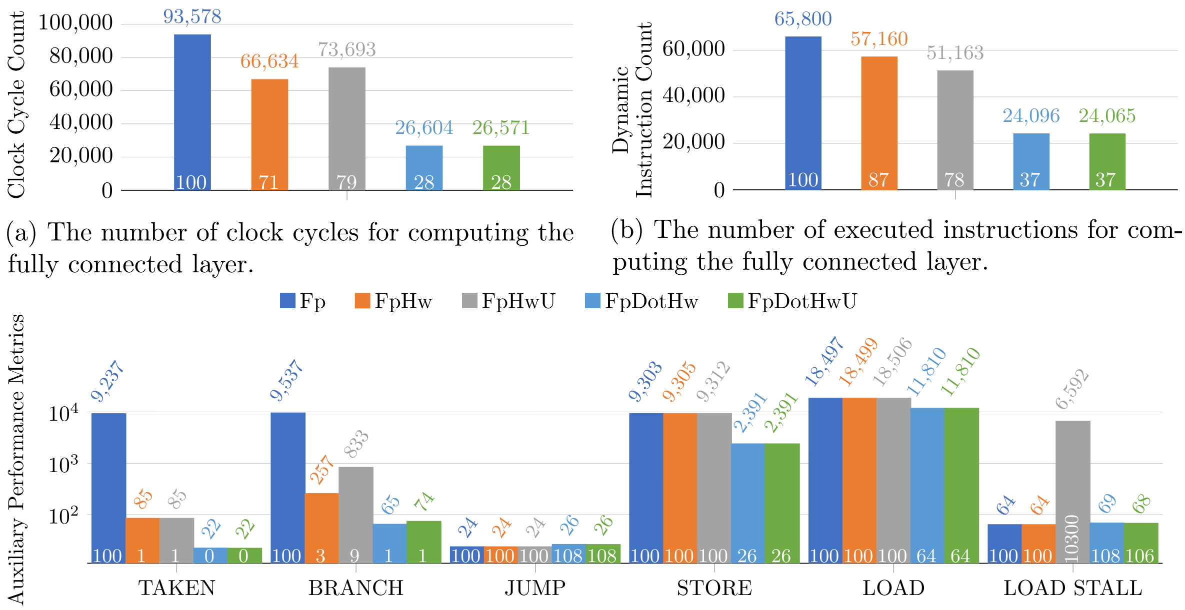


Figure 6.1: Code size comparison of the deep learning library.



(a) The number of clock cycles for computing the fully connected layer.

(b) The number of executed instructions for computing the fully connected layer.

(c) Auxiliary results for the fully connected layer.

Figure 6.2: Results for the fully connected layer. The numbers in the columns represent the relative percentages of a given metrics compared to the baseline Fp implementation.

6.4 Fully connected layer

Let us first look at the results for fully connected layers; it means computing a matrix-vector product, adding the bias, and applying the ReLU function. Most of the time is being used in matrix-vector multiplication. The reader should keep in mind that computing matrix-vector product is very memory intensive.

What we are measuring is computing the first layer of our demonstration application. The matrix-vector product consists of a matrix with a dimension 64×144 and a vector of 144 elements.

We ran the same code twenty times with different inputs and computed the average of the runs. The averages are displayed in Figure 6.2a and Figure 6.2b.

Figure 6.2a compares the number of clock cycles taken to compute the fully connected layer. Hardware loops alone contribute to a 29% reduction in clock cycles compared to the baseline (Fp). With the dot product unit included, a clock cycle reduction of 72% is achieved compared to the baseline. The result does make sense since we are replacing seven instructions (four multiplications and three additions) with just one. The result could be better if we could feed the data to the RI5CY core faster. Let us reason in the following way. It takes a single cycle to calculate a single dot product of two vectors of size four with our hardware. To load in the data for this dot product unit takes a minimum of eight clock cycles. As we operate on two vectors of size four and each aligned access takes one cycle, it means that we are using our unit only 11% of the time. This result is calculated in Equation (6.1).

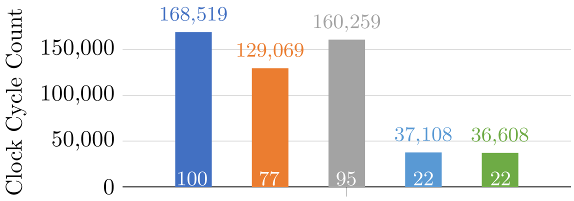
$$UTILIZATION = \frac{num.\ clock\ cycles\ dot\ prod.\ unit\ is\ used}{total\ clock\ cycles} = \frac{1}{8+1} \tag{6.1}$$

The situation gets a little better because we are reusing one vector several times (look at the algorithm we use in Figure 5.3). This means that the best possible utilization of our dot product unit is $1/(1 + 4) = 20\%$.

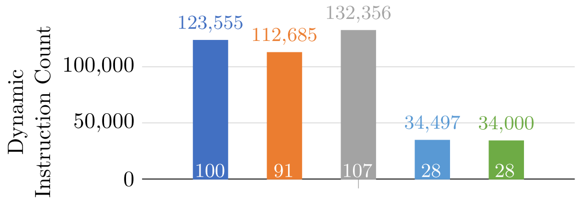
Using loop unrolling does not provide any benefit both in the implementation with the dot product unit and without it. In fact, it slows both the FpHwU, and the FpDotHwU implementation. The implementation using hardware loops and loop unrolling (FpHwU) achieved a 21% reduction in clock cycle count compared to the baseline, but a 29% reduction was achieved using only hardware loops (FpHw). The reason is that loop unrolling makes the caching characteristics of the code much worse (the cache is no longer big enough). Unrolling the loop also increases the number of branches significantly. Both effects are seen by looking at the number of load stalls and the number of branches for the (FpHwU) implementation in Figure 6.2c. The increased number of branches is the result of unrolling loops that contain conditional statements — namely dlReLU.

The dynamic instruction count comparison can be seen in Figure 6.2b. Hardware loops contribute a 13% reduction in dynamic instruction count compared to the baseline. The dynamic instruction count of the optimized inline assembly code for the dot product unit contributed a 63% reduction of the baseline dynamic instruction count. It is a predictable consequence of having one instruction that is equal to seven scalar instructions. Loop unrolling lowers the dynamic instruction count. The implementation using hardware loops and loop unrolling reduced the dynamic instruction count by 23% compared to the baseline; this is 10% better than using only hardware loops. Keep in mind that the FpHwU implementation of the neural network uses both hardware loops and loop unrolling. Unrolling the loop lowers the number of loop iterations in loops that are not implemented using hardware loops. All loops cannot be made into hardware loops due to limitations of the RI5CY core.

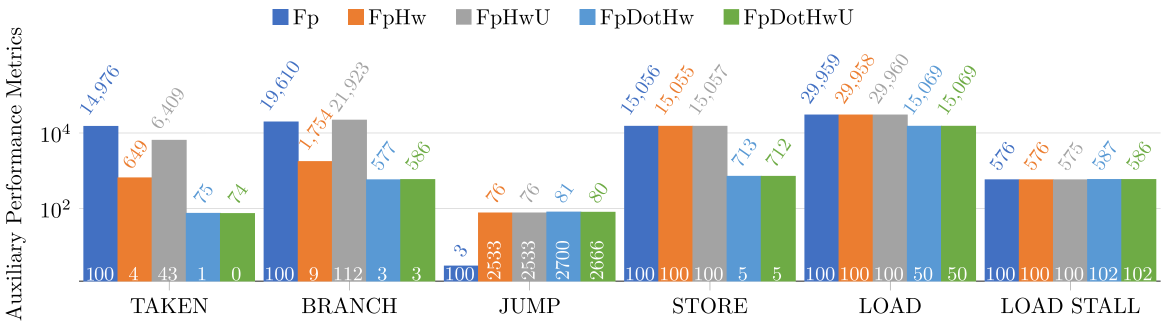
The results of other performance counters are shown in Figure 6.2c. We do not show the performance counters that were zero in all implementations. The version with the dot product unit seems to be mildly better or equal in every category. See Subsection 3.2.2 for an explanation of the performance counters.



(a) The number of clock cycles for computing the convolutional layer.



(b) The number of executed instructions for computing the convolutional layer.



(c) Auxiliary results for the convolutional layer.

Figure 6.3: Results for the convolutional layer. The numbers in the columns represent the relative percentages of a given metrics compared to the baseline Fp implementation.

Our inline assembly code for the dot product unit (FpDotHw) substantially lowers the number of branches that are made because we use a different algorithm. The naive algorithm has three for loops, only two of which can be made to use hardware loops due to limitations of the RI5CY core. It means that we are still executing branch instructions of the loops which cannot be made into hardware loops. Our algorithm also minimizes the number of loads and stores. As can be seen in Figure 6.2c, the number of loads and stores in the FpDotHw implementation is reduced by 36% and 74%, respectively, compared to the baseline.

6.5 Convolutional layer

In this section, we look at the cost of computing a convolution of a 28×28 input picture with a 5×5 filter. We also include the cost of computing the ReLU function of the resulting picture. We do this because we integrated the ReLU function into the convolution algorithm in order to optimize the code. This layer differs from the fully connected layer as it is not so constrained by memory bandwidth. This fact means that we can use our dot product unit more effectively. As in the case of the fully connected layer, we were not able to fetch data fast enough to utilize the unit entirely. It can be seen in Figure 6.3a that using our dot product unit (FpDotHw) contributes to a 78% reduction of the clock cycle count compared to the baseline implementation (Fp). Implementation using just hardware loops (FpHw) contributed only a 23% reduction. As it was the case in the fully connected layers, loop unrolling is not effective.

The dynamic instruction count is again substantially lower when using the dot product unit. This is a consequence of dot product instruction. With it, we replace seven instructions with one. In Figure 6.3b, we see that FpDotHw dynamic instruction count is reduced by 72% compared to the baseline dynamic instruction count. Adding only hardware loops (FpHw) again has a modest impact. The FpHw implementation dynamic instruction count is reduced only by 9% compared to the baseline. Loop unrolling (FpHwU) this time increases the dynamic instruction count by 7% compared to the baseline. The increase is because of the way the loops are unrolled significantly increases the number of branches. This effect is present in loops that have branches inside them, as, e.g., the function dlReLU. This issue also appears in the case with fully connected layers but becomes worse when we are running these functions over bigger two-dimensional arrays instead of vectors.

For our optimized assembly implementation of the convolutional layer, we load in the entire filter. It reduces the number of loads significantly as we only load the filter coefficients once. This strategy can be seen in Figure 6.3c. The FpDotHw and FpDotHwU implementations reduce the number of loads by 50% compared to the baseline (Fp). The number of stores is reduced by 95%. Hardware loops again decrease the number of branch instructions. The number of branches of the FpHw implementation is reduced by 91% compared to the baseline. Since we are using many small loops, hardware loops are a good idea. The FpDotHw implementation lowers the number of branches even further—by 97% compared to the baseline. The FpDotHw implementation has fewer branches, because the only branches the FpDotHw implementation has are the ones in the ReLU function while the FpHw function has additional branches for loops which are not hardware loops. As is the case with fully connected layers, the number of branches skyrockets when using loop unrolling. Although loop unrolling lowers branching in the convolution loop, it significantly increases it in the ReLU loop.

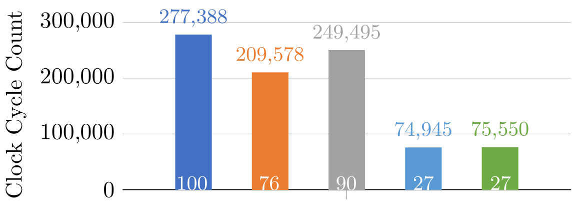
6.6 Full neural network

Lastly, we look at the results of running the full demonstration neural network. The full architecture of our network is shown in Figure 5.1. However, to quickly reiterate, we have the following layers:

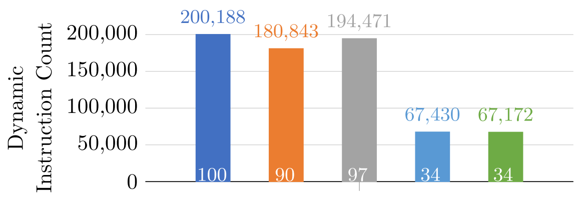
- convolutional layer + ReLU,
- maximum pooling layer,
- first fully connected layer + ReLU,
- second fully connected layer,
- softmax layer.

Figure 6.4a shows that for the FpDotHw implementation we have an average clock cycle count reduction of 73% compared to the baseline. This clock cycle count reduction is closer to the reduction for fully connected layers. It makes sense since we have more fully connected layers. Hardware loops alone (FpHw) reduce the clock cycle count by 24% compared to the baseline. If we ran our microcontroller at only 10 MHz, we could run a single inference of the neural network in 7.5 ms using the FpDotHw version.

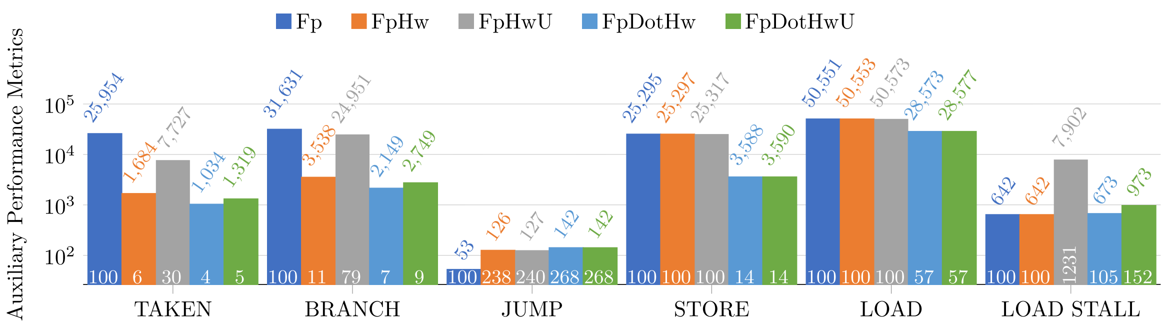
From Figure 6.4b, we see that the dynamic instruction count for the FpDotHw version is reduced by 66% compared to the baseline version. The FpHw version also reduces the dynamic instruction count, but only by 10%.



(a) The number of clock cycles for computing the entire neural network.



(b) The number of executed instructions for computing the entire neural network.



(c) Auxiliary results for the entire neural network.

Figure 6.4: Results for the entire neural network. The numbers in the columns represent the relative percentages of a given metrics compared to the baseline Fp implementation.

Figure 6.4c shows the average results of the auxiliary performance counters for the entire neural network. Again, we only show the performance counters that were nonzero. In general, we can say that using the dot product unit and the optimized inline assembly code (the FpDotHw implementation), we reduced the number of stores by 86%, and the number of loads by 43% compared against the baseline (Fp). Since neural networks are very data-intensive, this result is very promising. Branching characteristics of the program also got slightly better while the number of load stalls slightly increased.

6.7 Energy consumption

In this section, we show the energy consumption to run one inference. We derive these results from the ASIC synthesis power results and the clock cycle count results for the various implementation. Figure 6.5 shows the energy results of running our full neural network. The results are computed by taking the product between the power consumption of the particular implementation and the cycle delay of computing one pass of the full neural network. The first three implementations (Fp, FpHw, FpHwU) are without the dot product unit, and the last two include it.

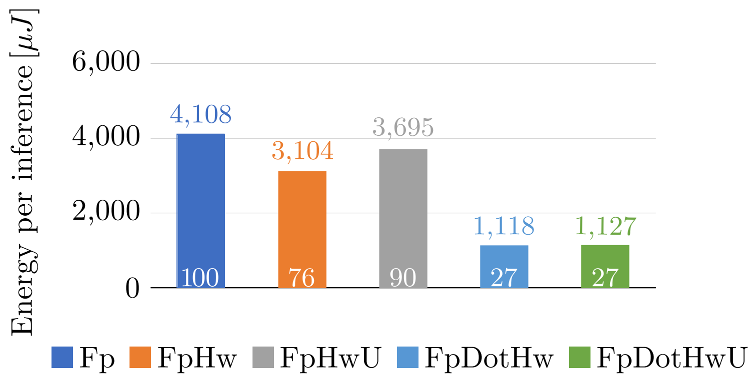


Figure 6.5: Energy consumption for one inference of our example neural network.

Adding the dot product unit (FpDotHw) reduced our energy consumption by 73% compared to the baseline. Hardware loops alone (FpHw) reduced the energy consumption by 24%, but if loop unrolling is also included (FpHwU), the energy consumption decrease is only 10% compared to the baseline.

Summary

In this chapter, we listed and analyzed our results. We first looked at the synthesis results. We saw that we have to pay a significant price in the area, but the power consumption stays within acceptable boundaries. The relatively significant increase in static power is a concern, even though many embedded devices operate mostly in standby mode. Next, we discussed the evaluation methodology we used. We list five different implementations we compare. The code size was listed next. We see that code size is within acceptable ranges, even for embedded devices. Following that, we listed and analyzed the results of running the different implementations of the neural network. The dot product unit decreases the clock cycle count by 73% compared to the baseline. The last and most important result showed that energy consumption did indeed decrease by using the dot product unit. In the next chapter, we compare our results with other similar implementations.

7.0 Comparison and conclusion

This chapter compares our system with similar research. We had some difficulties finding papers for a good comparison as ARM, the dominant supplier of processors for the microcontrollers, did not make public any technical results on Helium. It is a similar vector extension for ARM Cortex-M processors. Nevertheless, we will compare the features of Helium to our system based on the publicly available information. Our other comparison will be with [2]. In [2], they implemented a more complicated ISA extension based on the proposed RISC-V vector extension. They also ported TensorFlow Lite for that extension, which made it far easier to evaluate models. One idiosyncrasy of the [2] paper is that they use a functional simulator, which is not cycle-accurate. It ignores delays at memory transactions. Therefore, we can only compare instruction count results. Also, the results they provide for instruction count are for neural networks that we did not run. We will only provide a comparison between the ratios of the instruction count. Lastly, we conclude our results and show some prospects for future work on this topic.

7.1 Comparison of our design with other designs

7.1.1 Other work with RISC-V

In this section, we compare our work against [2]. The authors there implemented a customized ISA extension for RISC-V aimed at running machine learning applications. The extension is based on the proposed RISC-V vector (V) extension. However, they only added a subset of the vector extension. The instructions they added are listed in Table 7.1. They also ported TensorFlow Lite to this ISA. It allowed them to compile code for various models that can be found online. They tested the models on a functional simulator and Rocket core [61]. They only provide instruction count results. Figure 7.1 shows the software setup used in [2] to evaluate the benefit of added instructions.

They do not provide results for specific functions such as computing convolution or a matrix-

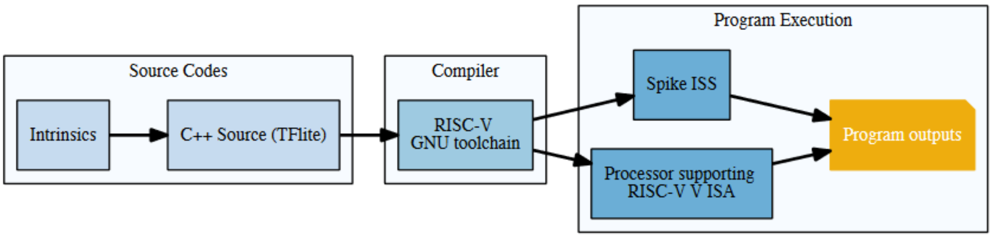


Figure 7.1: Software setup for evaluating instruction extensions [2].

vector product. Instead, they provide numbers for bigger networks that we did not port to our system. That is why we will only compare the relative lowering of the instruction count compared to there base implementation.

They report an average of 800% reduction in instruction count when using the ISA extension. Our reduction is closer to 300%. This is likely because they added vector load instructions. Our implementation requires four scalar loads instead of one vector load. Other functions can also be implemented more compactly with the added instructions. Instructions vls and vlx allow non-unit stride. They can be used for unaligned loads. Using this feature, we could replace lines 18, 19, 20, and 21 in Algorithm 5.2 with just one instruction.

Element-wise maximum/minimum instructions could be used for an optimized implementation of the ReLU function. To be specific, this could be achieved with vfmax instruction. Our code looks at each element one by one. However, with the vfmax instruction, we can compare one vector at a time. We have to fill one source register of vfmax with data, and the other source register should be all zero.

7.1.2 Armv8.1-M architecture and Helium

In this subsection, we show that ARM is pursuing similar techniques to bring machine learning to microcontrollers. For this, they developed ISA extensions collectively called as the Armv8.1-M architecture. Armv8.1-M architecture is an enhancement of ARM architecture for the microcontroller class processors (Cortex-M). Note that there are no numbers available for ARMv8.1-M, so our comparison will not be quantitative. The main addition of ARMv8.1-M architectures is a vector extension called Helium. Other additions include [4]:

- low overhead branch extension,
- instructions for half-precision floating-point support,
- instruction set enhancement for TrustZone management for a floating-point unit,
- new memory attribute in the memory protection unit,

Table 7.1: The subset of RISC-V Vector ISA extension implemented in [2].

Inst. type	Instructions	Function
Memory access	vls, vlx, vss, vsx	Load and store vector from memory into a destination vector register. With unit/const stride or with indexed stride.
Simple arithmetic	vadd, vmul, vfadd, vfmul	Add/Multiply values in two source vector registers and write the result to the destination vector register.
MAC arithmetic	vmadd, vfmadd	Multiply values in two source vector register and add third source vector register. The result is stored in the destination vector register.
Max/min	vmax, vmin, vfmax, vfmin	Element-wise max/min of values into two source vector registers. The result is stored in the destination vector register.
Splat/broadcast	vsplat, vbcastx, vbcastf	Splat an element of a source vector register to the destination vector register. Broadcast the value in source scalar register to destination vector register.
Reductions	vredsum, vredmin, vredmax, vfredsum	Reduction of a source vector register based on sum/max/min, broadcast, and store the result to the destination vector register.

- debug enhancements,
- reliability, availability, and serviceability extension.

The Helium extension, coupled with the low overhead loop extensions, makes a system very much like ours.

7.1.2.1 Helium

Both our system, as well as Helium, use a 128-bit vector. Additionally, as it is aimed at microcontrollers, it reuses the floating-point register file just as our system to save area and power. Helium is a more comprehensive extension. It adds over 150 new instructions, including vector load instructions, lane predication, complex math, and scatter-gather memory accesses [4]. They define vector lane width. It is the number of elements in a vector. For example, we have a 128-bit vector length. If we operate on 32-bit data types, vector lane width is four.

Vector load instructions make for a much more compact code. [2] shows that with similar extensions, they got on average 800% smaller instruction count, which is much better than our 300% reduction. Additionally, scatter-gather memory access instructions allow the software to handle arbitrary memory access patterns [4]. They can speed up non-sequential access to data elements in an array.

Lane predication is a mechanism for conditional execution of the lanes in the vector. To enable this feature, they added a new special-purpose register to hold the condition for each lane. This predication condition inside this register is updated by vector operations, such as vector compare instructions. After the condition flags are set, we can then set the conditional execution of each lane. It can be useful to implement functions that involve a lot of condition checking. A specific example is applying the ReLU nonlinearity to a vector or matrix.

7.1.2.2 Low Overhead Branch Extension

This extension is very similar to the hardware loop extension used by the RI5CY core (see Subsection 3.2.1). Listing 7.1 shows how WLS instruction can be used to create a loop that will be executed ten times. We first set register R0 to 10. Then we start the loop execution with the function WLS. The loop copies data from one part of the memory to another.

In addition to WLS, there is also a do-loop-start (DLS) instruction [4]. This instruction executes the loop body at least once. It makes very natural modeling of high-level programming structures like for loops and do-while loops.

Listing 7.1: Low Overhead Branch Extension by ARM [4].

```
    MOVS R0, #10 ; loop 10 times
    WLS  LR, R0, loopend_1; while loop start
loopstart_1
    LDR  R1, [R2], #4 ; Memory copy
    STR  R1, [R3], #4 ; Memory copy
    LE   LR, loopstart_1 ; loop end
loopend_1
    B label_x ; branch to somewhere else
```

There also exist instructions which can be used for loop tail prediction if we are processing an element of size N, where N is not a multiple of the vector lane width. This feature enables us to process just the remaining elements using conditional execution. For example, suppose we have a vector-vector dot product, and that both vectors have 15 elements. We could then issue 3 standard dot product instructions of size 4 and one of size 3. These types of instructions use bits of the instruction to signify, which parts of the vector should be executed.

7.2 Summary

We started by motivating the reader for the use of neural network algorithms in edge devices. We showed that there is a gap of capability in the massive embedded devices market. Such applications could benefit from using neural network algorithms, but the traditional view was that microcontrollers lack performance to run neural networks. We then presented the hardware used. The open-source microcontroller called PULPino was the basis, and we extended it with a dot product unit for floating-point numbers. Next, we presented a system for evaluating microcontroller performance by emulating it on an FPGA. We used a Zynq-7000 system, which has an ARM Cortex-A9 core and an FPGA part. On the ARM Cortex-A9, we ran Linux and a user space application. We used the user space application to control, program, and monitor the emulated microcontroller running our neural network.

To emulate sensor data, we feed data to the microcontroller through the AXI bus. This approach allowed us to send arbitrary data to the microcontroller. Using this, we explored the effect of using various instruction set extensions to boost the capability of our microcontroller to run neural network algorithms. Specifically, we looked at hardware loops and a simple dot product unit. We found that dot product units or other more general SIMD units can offer a significant clock cycle count decrease when running neural networks. Our dot product unit that used a vector length of four decreased the clock cycle count by 72%. Such units come at a cost in area and energy consumption. We also looked at hardware loops. They are used to minimize the overhead of loops with small bodies. Such structures are ubiquitous in neural networks. We saw that using just hardware loops we can reduce the clock cycle count by around 24%. Hardware loops also add only a tiny die area.

Additionally, compilers can easily be modified to use hardware loops where appropriate automatically. This is less true for vector instructions, which mostly must be handwritten to be useful. We determined that loop unrolling was not an effective solution for our system. As small caches of our system quickly became overwhelmed, and that triggered more accesses to memory. The bodies of most loops are small enough to fit entirely into the cache. Additionally, the load-store unit of the RI5CY core is hardware loop aware and knows if the instructions should be kept in cache or not. Lastly, we looked at existing designs aimed at the embedded market. We show that ARM, the dominant microcontroller core supplier, is pursuing similar but broader instruction set extensions. Similar extensions are also in work for the RISC-V instruction set.

7.3 Conclusion

Our work showed that it is possible to run limited neural networks even on low power systems like microcontrollers. Dot product extension can help boost the capability of the microcontroller by 370% if extra performance is needed. Because of memory issues, convolutional neural networks are more suited to run on microcontrollers for two reasons. The first reason is that one convolutional layer takes up less memory space, which is usually very scarce on microcontrollers. The second reason is because of the limited memory bandwidth. In the case of the fully connected layer, we could not fully utilize the dot product unit because we were waiting on memory. Additionally, hardware loops alone reduce the clock cycle count by 26% compared to the baseline, so it pays to include them in the ISA. The results of this Master's thesis will be published in [62].

7.4 Future work

A significant factor slowing down the dot product unit is memory bandwidth. This fact is especially actual when running fully connected layers. One potential solution for such a case is using lower precision arithmetic. For example, using a 16-bit floating-point format (also called half-float) can still deliver satisfiable accuracy [13] of the neural network and simultaneously halve the memory footprint and bandwidth requirements. Using lower precision data can alleviate the memory bottleneck and increases the utilization of the dot product unit.

Another way forward would be to replace the floating-point format all together with other formats. Many researchers have explored using just pure integer arithmetic. However, for some algorithms, the dynamic range of integer arithmetic is a problem. Another option is posit arithmetic [63] [64]. Posit arithmetic is a replacement for the IEEE-754 floating-point format. It provides superior accuracy and dynamic range when using the same number of bits.

Bibliography

[1] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017. DOI: 10.1109/TVLSI.2017.2654506.

[2] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using TensorFlow Lite on RISC-V", *CARRV*, 2019. DOI: https://doi.org/10.1145/1122445.1122456.

[3] elinux.org, *Device tree usage*, https://elinux.org/, Apr. 29, 2019. [Online]. Available: https://elinux.org/Device_Tree_Usage (visited on 08/10/2019).

[4] J. Yiu, "Introduction to armv8.1-m architecture", ARM Ltd., Tech. Rep., Feb. 2019.

[5] S. Kodali, P. Hansen, N. Mulholland, P. Whatmough, D. Brooks, and G.-Y. Wei, "Applications of deep neural networks for ultra low power IoT", in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017. DOI: 10.1109/ICCD.2017.102.

[6] Y. Chen, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks", *CoRR*, vol. abs/1807.07928, 2018. arXiv: 1807.07928. [Online]. Available: http://arxiv.org/abs/1807.07928.

[7] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network", *CoRR*, vol. abs/1602.01528, 2016. arXiv: 1602.01528. [Online]. Available: http://arxiv.org/abs/1602.01528.

[8] D. K. Mandal, J. Sankaran, A. Gupta, K. Castille, S. Gondkar, S. Kamath, P. Sundar, and A. Phipps, "An embedded vision engine (EVE) or automotive vision processing", in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014. DOI: 10.1109/ISCAS.2014.6865062.

[9] A. Ltd., *ARM machine learning processor NPU*, English, Arm Ltd., 2019, 2 pp., published.

[10] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT", in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2018, pp. 1–4. DOI: 10.1109/ASAP.2018.8445101.

[11] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: A hardware accelerator IP for neural network inference", *CoRR*, vol. abs/1807.03010, 2018. arXiv: 1807.03010. [Online]. Available: http://arxiv.org/abs/1807.03010.

[12] I. A. GmbH, *State of the IoT 2018: Number of IoT devices now at 7b – market accelerating*, Aug. 8, 2018. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ (visited on 08/20/2019).

[13] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding", in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. [Online]. Available: http://arxiv.org/abs/1510.00149.

[14] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough, "SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers", 2019. [Online]. Available: https://arxiv.org/abs/1905.12107.

[15] M. Horowitz, "Computing's energy problem (and what we can do about it)", vol. 57, Feb. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[16] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using TensorFlow Lite on RISC-V", *Third Workshop on Computer Architecture Research with RISC-V*, no. 22, p. 6, Oct. 2019. DOI: 10.1145/1122445.1122456.

[17] E. Torti, A. Fontanella, M. Musci, N. Blago, D. Pau, F. Leporati, and M. Piastra, "Embedded real-time fall detection with deep learning on wearable devices", in *2018 21st Euromicro Conference on Digital System Design (DSD)*, Aug. 2018, pp. 405–412. DOI: 10.1109/DSD.2018.00075.

[18] W. H. Organization, *WHO global report on falls prevention in older age*, 2008.

[19] M. Hügle, S. Heller, M. Watter, M. Blum, F. Manzouri, M. Dümpelmann, A. Schulze-Bonhage, P. Woias, and J. Boedecker, "Early seizure detection with an energy-efficient convolutional neural network on an implantable microcontroller", 2018. arXiv: 1806.04549. [Online]. Available: https://arxiv.org/abs/1806.04549.

[20] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello Edge: Keyword spotting on microcontrollers", *CoRR*, vol. abs/1711.07128, 2017. arXiv: 1711.07128. [Online]. Available: http://arxiv.org/abs/1711.07128.

[21] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "Ultra low power deep-learning-powered autonomous nano drones", *CoRR*, vol. abs/1805.01831, 2018. arXiv: 1805.01831. [Online]. Available: http://arxiv.org/abs/1805.01831.

[22] T. Borah, K. Sarma, and P. Talukdar, "Biometric identification system using neuro and fuzzy computational approaches", in. Dec. 2015, pp. 335–368. DOI: 10.4018/978-1-4666-8654-0.ch016.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks", in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[24] X. Inc., *Ultrascale architecture and product data sheet: Overview*, English, version 3.9, Xilinx Inc., Jun. 2019, 48 pp., published.

[25] P. Kallstrom, *Field-programmable gate array*, https://en.wikipedia.org/wiki/Field-programmable_gate_array, Aug. 20, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array (visited on 08/13/2019).

[26] P. Coussy, M. Meredith, A. Takach, and D. D. Gajski, "An introduction to high-level synthesis", *IEEE Design & Test of Computers*, vol. 26, no. 04, pp. 8–17, Jul. 2009. DOI: 10.1109/MDT.2009.69.

[27] A. Ltd., "AMBA AXI and ACE protocol specification", ARM Ltd., Tech. Rep., Jun. 2011.

[28] ——, "AMBA APB protocol specification", ARM Ltd., Tech. Rep., Apr. 2010.

[29] T. L. I. Project, *Kernel definition*, Aug. 6, 2019. [Online]. Available: http://www.linfo.org/kernel.html (visited on 08/06/2019).

[30] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, Nov. 2005, ISBN: 0-596-00565-2.

[31] R. Shingledecker, *Tiny Core Linux*, http://www.tinycorelinux.net/, Dec. 1, 2008. [Online]. Available: http://www.tinycorelinux.net/ (visited on 08/10/2019).

[32] elinux.org, *Device tree history*, https://elinux.org/, Jun. 9, 2016. [Online]. Available: https://elinux.org/Device_tree_history (visited on 08/10/2019).

[33] devicetree.org, *Devicetree*, https://www.devicetree.org/, Aug. 6, 2019. [Online]. Available: https://www.devicetree.org/ (visited on 08/06/2019).

[34] H.-J. Koch and Linutronix, *The Userspace I/O howto – the Linux kernel documentation*, Dec. 11, 2016. [Online]. Available: https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html (visited on 08/06/2019).

[35] R. Foundation, *RISC-V foundation | instruction set architecture (ISA)*. [Online]. Available: https://riscv.org/ (visited on 07/27/2019).

[36] A. Waterman, "Design of the RISC-V instruction set architecture", University of California at Berkeley, Tech. Rep., Jan. 2016.

[37] T. P. team, *PULP platform*, https://www.pulp-platform.org/index.html, Aug. 4, 2019. [Online]. Available: https://www.pulp-platform.org/index.html (visited on 08/04/2019).

[38] A. Traber and M. Gautschi, "PULPino: Datasheet", ETH Zürich, Tech. Rep., Jun. 2017.

[39] A. Traber, M. G. Pasquale, and D. Schiavone, *RI5CY: User manual*, English, version 2.0, Micrel Lab and Multitherman Lab, University of Bologna, Italy, Mar. 2018, 60 pp., published.

[40] A. Baluni, F. Merchant, S. K. Nandy, and S. Balakrishnan, "A fully pipelined modular multiple precision floating point multiplier with vector support", in *International Symposium on Electronic System Design, ISED 2011, Kochi, Kerala, India, December 19-21, 2011*, IEEE Computer Society, 2011, pp. 45–50. DOI: 10.1109/ISED.2011.14. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ISED.2011.14.

[41] F. Merchant, A. Chattopadhyay, S. Raha, S. K. Nandy, and R. Narayan, "Accelerating BLAS and LAPACK via efficient floating point architecture design", *Parallel Processing Letters*, vol. 27, no. 3-4, 1750006:1–1750006:17, 2017. DOI: 10.1142/S0129626417500062. [Online]. Available: https://doi.org/10.1142/S0129626417500062.

[42] F. Merchant, T. Vatwani, A. Chattopadhyay, S. Raha, S. K. Nandy, R. Narayan, and R. Leupers, "Applying modified householder transform to kalman filter", in *32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems, VLSID 2019, Delhi, India, January 5-9, 2019*, IEEE, 2019, pp. 431–436. DOI: 10.1109/VLSID.2019.00092. [Online]. Available: https://doi.org/10.1109/VLSID.2019.00092.

[43] ——, “A systematic approach for acceleration of matrix-vector operations in CGRA through algorithm-architecture co-design”, in *32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems, VLSID 2019, Delhi, India, January 5-9, 2019*, IEEE, 2019, pp. 64–69. DOI: 10.1109/VLSID.2019.00030. [Online]. Available: https://doi.org/10.1109/VLSID.2019.00030.

[44] F. Merchant, T. Vatwani, A. Chattopadhyay, S. Raha, S. K. Nandy, and R. Narayan, “Efficient realization of householder transform through algorithm-architecture co-design for acceleration of QR factorization”, *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1707–1720, 2018. DOI: 10.1109/TPDS.2018.2803820. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2803820.

[45] M. Mahadurkar, F. Merchant, A. Maity, K. Vatwani, I. Munje, N. Gopalan, S. K. Nandy, and R. Narayan, “Co-exploration of NLA kernels and specification of compute elements in distributed memory cgras”, in *XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14-17, 2014*, IEEE, 2014, pp. 225–232. DOI: 10.1109/SAMOS.2014.6893215. [Online]. Available: https://doi.org/10.1109/SAMOS.2014.6893215.

[46] F. Merchant, A. Chattopadhyay, G. Garga, S. K. Nandy, R. Narayan, and N. Gopalan, “Efficient QR decomposition using low complexity column-wise givens rotation (CGR)”, in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, India, January 5-9, 2014*, IEEE Computer Society, 2014, pp. 258–263. DOI: 10.1109/VLSID.2014.51. [Online]. Available: https://doi.org/10.1109/VLSID.2014.51.

[47] Z. E. Rákossy, F. Merchant, A. A. Aponte, S. K. Nandy, and A. Chattopadhyay, “Scalable and energy-efficient reconfigurable accelerator for column-wise givens rotation”, in *22nd International Conference on Very Large Scale Integration, VLSI-SoC, Playa del Carmen, Mexico, October 6-8, 2014*, L. Garcia, Ed., IEEE, 2014, pp. 1–6. DOI: 10.1109/VLSI-SoC.2014.7004166. [Online]. Available: https://doi.org/10.1109/VLSI-SoC.2014.7004166.

[48] F. Merchant, A. Maity, M. Mahadurkar, K. Vatwani, I. Munje, M. K. C, N. Sivanandan, N. Gopalan, S. Raha, S. K. Nandy, and R. Narayan, “Micro-architectural enhancements in distributed memory CGRAs for LU and QR factorizations”, in *28th International Conference on VLSI Design, VLSID 2015, Bangalore, India, January 3-7, 2015*, IEEE Computer Society, 2015, pp. 153–158. DOI: 10.1109/VLSID.2015.31. [Online]. Available: https://doi.org/10.1109/VLSID.2015.31.

[49] S. Das, K. T. Madhu, M. Krishna, N. Sivanandan, F. Merchant, S. Natarajan, I. Biswas, A. Pulli, S. K. Nandy, and R. Narayan, "A framework for post-silicon realization of arbitrary instruction extensions on reconfigurable data-paths", *J. Syst. Archit.*, vol. 60, no. 7, pp. 592–614, 2014. DOI: 10.1016/j.sysarc.2014.06.002. [Online]. Available: https://doi.org/10.1016/j.sysarc.2014.06.002.

[50] X. Inc., *Zynq-7000 SoC*, https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, Jul. 4, 2019. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (visited on 07/04/2019).

[51] A. Inc., *ZedBoard | ZedBoard*, http://zedboard.org/, Aug. 6, 2019. [Online]. Available: http://zedboard.org/product/zedboard (visited on 08/06/2019).

[52] X. Inc., *Xilinx - adaptable. intelligent.* https://www.xilinx.com, Aug. 7, 2019. [Online]. Available: https://www.xilinx.com (visited on 08/07/2019).

[53] Y. LeCun, *The MNIST database of handwritten digits*, Jul. 2019. [Online]. Available: http://yann.lecun.com/exdb/mnist/ (visited on 07/29/2019).

[54] A. Kusupati†, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network", in *32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada*, 2018.

[55] G. LLC, *Micro speech*, https://github.com/tensorflow/, Jul. 2019. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/experimental/micro/examples/micro_speech (visited on 07/29/2019).

[56] Z. Q. Lin, A. G. Chung, and A. Wong, "Edgespeechnets: Highly efficient deep neural networks for speech recognition on the edge", in *32nd Conference on Neural Information Processing Systems (NIPS 2018)*, 2018.

[57] G. LLC, *Micro vision*, https://github.com/tensorflow/, Jul. 2019. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/experimental/micro/examples/micro_vision (visited on 07/29/2019).

[58] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size", *CoRR*, vol. abs/1602.07360, 2016. arXiv: 1602.07360. [Online]. Available: http://arxiv.org/abs/1602.07360.

[59] A. Sharma and H. Foroosh, “Slim-CNN: A light-weight CNN for face attribute prediction”, p. 11, Jul. 2019.

[60] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, “Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation”, *CoRR*, vol. abs/1801.04381, 2018. arXiv: 1801.04381. [Online]. Available: http://arxiv.org/abs/1801.04381.

[61] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The rocket chip generator”, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[62] J. Vreča, K. J. X. Sturm, E. Gungl, F. Merchant, P. Bientinesi, and Z. Brezočnik, “Accelerating deep learning inference in constrained embedded devices using hardware loops and a dot product unit”, *To be published in IEEE Access*, 2020.

[63] J. L. Gustafson and I. Yonemoto, “Beating floating point at its own game: Posit arithmetic”, 2017. DOI: 10.14529/jsfi170206.

[64] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, “Parameterized posit arithmetic hardware generator”, in *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*, IEEE Computer Society, 2018, pp. 334–341. DOI: 10.1109/ICCD.2018.00057. [Online]. Available: https://doi.org/10.1109/ICCD.2018.00057.