Deep Neural Network on the Versat Reconfigurable Processor

João Pedro Costa Luís Cardoso

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. José João Henriques Teixeira de Sousa

September 2022

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Dedicated to my fiancée Matilde Moreira and soon to be born son, Francisco.

Acknowledgments

I want to thank my supervisor and Professor José Teixeira de Sousa for his eternal patience with me finishing this dissertation and the opportunity to work on the Versat CGRA. I would also like to acknowledge my friends and my parents who are always there for me and a special mention to my fiancée who has supported me all the way through my Bachelor and Masters and has pushed me to finally finish this document.

Resumo

Esta tese apresenta uma solução para simular o Deep Versat, uma CGRA que é acupulada de um processador RISC-V. Tambem é apresentada nesta tese ferramentas para o Deep Versat correr Redes Neuronais Convolucionais. Estas cargas de trabalho são usadas em diversos algoritmos de Inteligencia Arteficial como a deteção de objetos em imagens. As vantagens da ferramenta são várias. Primeiro, a escrita das configurações do Versat é preciso conhecimento da arquitetura a nivel detalhado e das suas APIs. Segundo, a escrita de algoritmos complexos para o Versat é preciso muitas horas de desenvolvimento e mais outras quantas para testar em hardware, ou seja o custo de usar o Versat baixa consideravelmente e a performance é otimizada á configuração do Versat escolhida podendo testar centenas de configurações de Hardware para otimizar a performance de uma rede em especifico.

Palavras-chave: CGRA, Versat, Darknet, Redes Neuronais

Abstract

This thesis presents a solution to simulate Deep Versat, a CGRA, which is coupled to a RISC-V CPU. It also presented in this thesis the tools for Deep Versat to run any Convolutional Neural Network with any configuration of datapaths. These workloads are used in Machine Learning algorithms with object detention in images. The tool has several advantages. Firstly, in the configuration writing to the Registers, there's a need for a high degree of knowledge of the architecture of the CGRA and its software APIs. Secondly, the writing of complex algorithms on this hardware needs long hours of debugging and development, meaning by the use of the tools presented in this thesis, the development time can be reduced and performance can be automatically optimized. Finally, the tools can be adapted to changes in the hardware by changing a few software functions at most.

Keywords: CGRA, Versat, Darknet, Convolutional Neural Networks, Deep Neural Networks

Contents

List of Tables

List of Figures

List of Acronyms

AGU Address Generation Unit

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASIP Application Specific Instruction Set Processor

BRAM Block Random Access Memory

CGRA Coarse-Grain Reconfigurable Array

CM Configuration Module

CNN Convolutional Neural Network

CPI Cycles Per Instruction

CPU Central Processing Unit

DE Data Engine

DMA Direct Memory Access

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

FU Functional Unit

HDL Hardware Description Language

ISA Instruction Set Architecture

LED Light Emitting Diode

LUT Lookup Table

RTL Register-Transfer Level

UART Universal Asynchronous Receiver-Transmitter

UUT Unit Under Test

VCD Value Change Dump

VPI Verilog Procedural Interface

Chapter 1

Introduction

In this report, the problem of accelerating the execution of Deep Neural Networks (DNNs) using Coarse GRained Reconfigurable Arrays (CGRAs) is studied, with special emphasis on compiling a DNN description into code that runs on CPU/CGRA system. The Deep Versat Architecture [1] CGRA will be used as an implementation tool in this work.

1.1 Problem

Neural Networks have been an object of study since the 1940's, but until the beginning of this decade their applications were limited and did not play a major role in computer vision conferences. With its meteoric rise in research, several solutions to accelerate this algorithm have appeared, from Field Programmable Gate Arrays (FPGA) to Application Specific Integrated Circuits (ASIC) implementations.

Convolutional Neural Networks (CNNs) are a particular kind of DNN where the output values of the neurons in one layer are convolved with a kernel to produce the input values of the neurons of the next layer. This algorithm is compute bound, that is, its performance depends on how fast it can do certain calculations, and depend less on the memory access time. Namely the convolutional layers take approximately 90$\%$ of the computation time.

The acceleration of these workloads is a matter of importance for today's applications such as image processing for object recognition or simply to enhance certain images. Other uses like instant translation and virtual assistants are applications of neural networks and their acceleration is of vital importance to bring them into Internet of Things.

A suitable circuit to accelerate DNNs in hardware is the CGRA. A CGRA is a collection of Functional Units and memories with programmable interconnections in order to form computational datapaths. A CGRA can be implemented in both FPGAs and ASICs. CGRAs can be reconfigured much faster than FPGAs, as they have much less configuration bits. If reconfiguration is done at runtime, CGRAs add temporal scalability to the spacial scalability that characterize FPGAs. Moreover, partial reconfiguration is much easier to do in CGRAs compared to FPGAs which further speeds up reconfiguration time. Another advantage of CGRAs is the fact that they can be programmed entirely in software, contrasting with the large development time of customized Intellectual Property (IP) blocks. The Coarse Grain Reconfigurable Arrays (CGRA) is a midway acceleration solution between FPGAs, which are flexible but large, power hungry and difficult to reprogram, and ASICs, which are fast but generally not programmable.

However, mapping a specific DNN to a CGRA requires knowledge of its architecture, latencies and register configurations, which may become a lengthy process, especially if the user wants to explore the design space for several DNN configurations. An automatic compiler that can map a standard DNN description into CPU/CGRA code would dramatically decrease time to market of its users. Currently there are equivalent tools for CPUs and GPUs and even for FPGAS.

1.2 Solution

The proposed solution is a compiler that takes a configuration file from a neural network framework like Caffe or Darknet. This new tool inputs the parameters of Deep Versat, such as the number of layers and functional units, and produces the C code needed for the Versat runs. This code is run on the RISC-V picorv32 [2] CPU controller that has Deep Versat as a peripheral.

1.3 Report Outline

This report is composed of 4 more chapters. In the second chapter, the state-of-the-art of neural networks and the difficulties accelerating them is described. In the third chapter, the Deep Versat architecture and how to program it is explained. In the fourth chapter, CNN compiler techniques are explored. Finally, the last chapter contains the proposed solution and the plan for its execution.

Chapter 2

Background

2.1 Deep Neural Networks

A Neural Network (NN) is an interconnected group of nodes that follow a computational model that propagates data forward while processing. The earliest NNs were proposed by McCulloh and Pitts [3], in which a neuron has a linear part, based on aggregation of data and then a non-linear part called the activation function, which is applied to the aggregate sum. The issue with using only one neuron is that it is not able to be used in non-linear separable problems. By aggregating several neurons in layers and the input of each neuron as in figure 2.1 being based on the previous layers, that problem can be eliminated.

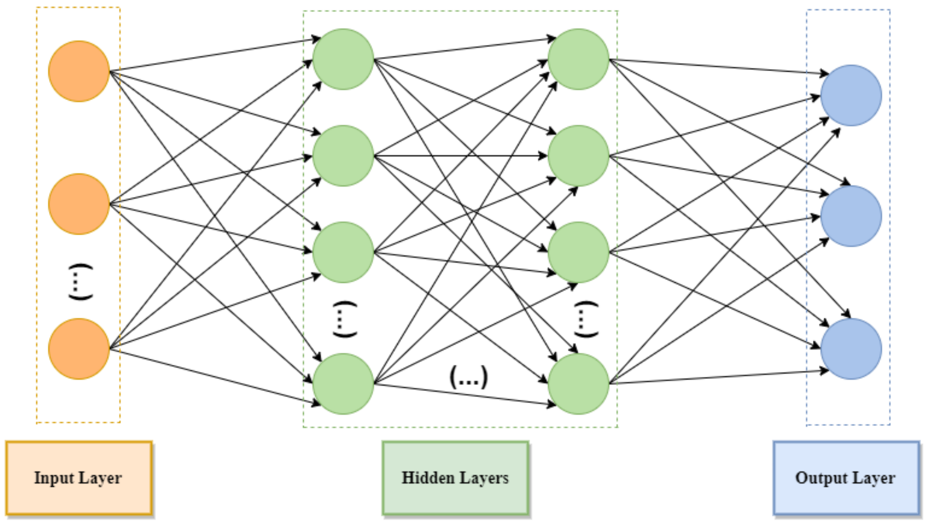

Figure 2.1: Deep Neural Network Structure

Each input to a neuron contributes differently to the output. The share is dependent on the weight value. These are obtained by training the network through various techniques, one of which is called Deep Supervised Learning [4] . For a certain input, there is an expected output and the real output of the

NN. Then the loss function (the difference) is calculated and the weight values are iteratively modified for improving the outputs of the NN.

A Deep Neural Network (DNN) is a Neural Network that uses this approach for learning. It has multiple hidden layers and it can model complex non-linear relationships. If the activation function is non polynomial, it satisfies the Universal approximation problem [5].

One of the limitations of traditional NNs is the complexity of layer interconnections. Using as example the hand digit recognition problem and MNIST data set, composed of 28x28 grayscaled images [6], in a traditional fully connected NN, a neuron from the second layer would have 28x28 weights. That is 3.136 kiloBytes per neuron of weight values while using 32-bit floating-point numbers (FP32). When building a more complex network for image recognition, the computationally complexity grows quadratically with the number of neuros per layer.

2.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a class of DNNs used in Image and Video recognition due to their shift invariance characteristic. They were first proposed in the 1980s but it was not until 2012 with AlexNet [7] that CNNs really took off. Fundamentally, CNNs are a regularized version of Multilayer Perceptrons (MLP). These networks fix the complexity issue discussed as each neuron is only connected to a few neurons of the previous layer.

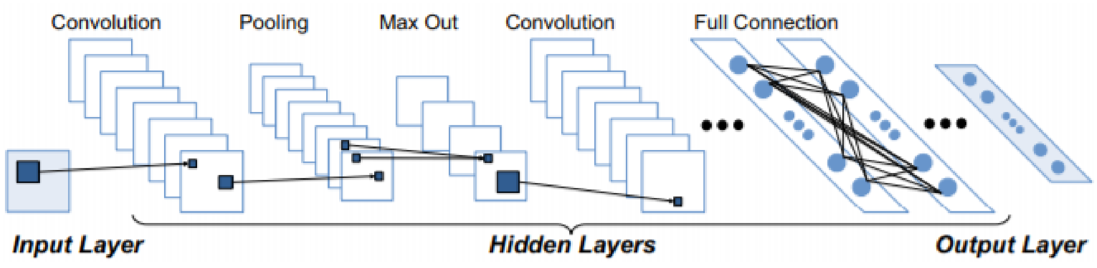

Figure 2.2: CNN architecture example, taken from [8]

2.1.1.1 Architecture Overview

2.1.1.1.1 Convolutional Layer

In a typical CNN not all layers are convolutional, but the convolutional layers are the most compute intensive ones. CNNs take input images with 3 dimensions (width, height and color space); for the following convolutional layers 3D arrays are used (width, height and number of channels). For the earlier example of the MNIST data set, the input would have dimensions 28x28x1 as it is a 2D image in grayscale.

To compute a neuron in the next layer we use the convolution equation 2.1 aided by Figure 2.3.

$$x_j^{l+1} = \delta(\sum_{i\epsilon M_j} x_i^l * k_{ij}^{l+1} + b_j^{l+1}) \tag{2.1}$$

where x_j^{l+1} is the output, δ is the activation function, which depends on the architecture, x_i^l is the input of the convolution layer, k_{ij}^{l+1} is the kernel of said layer which is obtained by training the network, and b_j^{l+1} is the bias.

Thus an output neuron depends only on a small region of the input which is called the local receptive field.

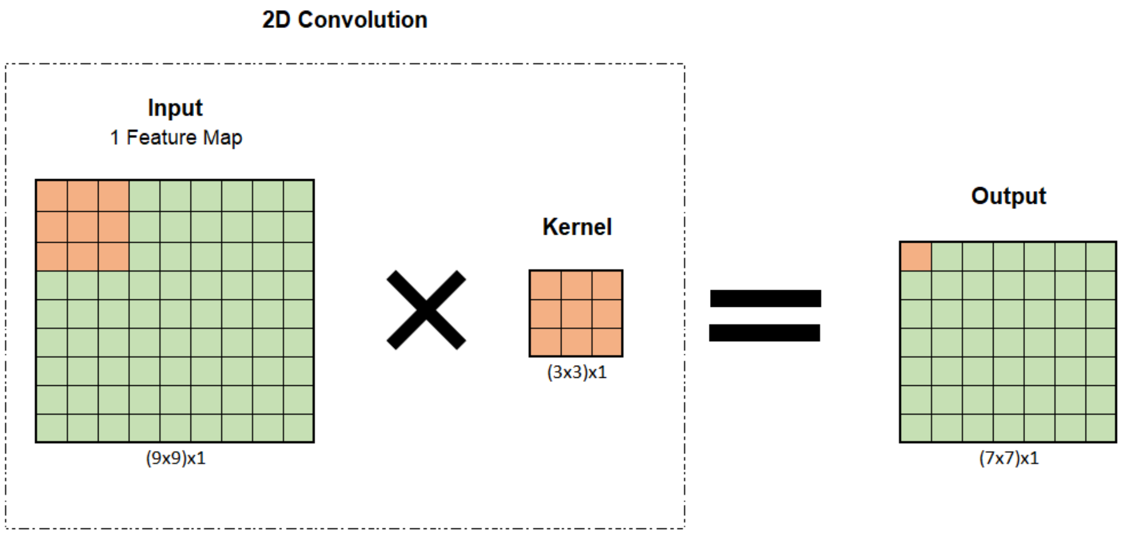

Figure 2.3: 2D convolution with stride = 1 and without zero padding

The output's dimensions depend on several parameters of the convolution such as zero-padding and stride. The former means to add zeros around the edges of the input matrix. The latter means the step used for the convolution, if the value is e.g 2, it will skip a pixel each iteration of the convolution. Equation 2.2 can be used to calculate the output size.

$$n^{l+1} = \frac{n^l - b^l + 2 \times p}{s} + 1 \tag{2.2}$$

where n is the width/height of the input of layer l, b is the width/height of the kernel, p is zero-padding while s is the stride.

The number of channels of the output is equal to the number of filters in the convolutional layer.

2.1.1.1.2 Pooling Layer

The MaxPool or AvgPool are layers used in Convolutional Neural Networks to downsampling the feature maps to make the output maps less sensitive to the location of the features.

Maximum Pooling or MaxPool, like it is suggested in its name groups $n*n$ points and outputs the pixel with highest value. The output will have its size lowered by n times. The Average Pooling or AvgPool, instead takes all of the input points and calculates the average. Downsampling can also be achieved by using convolutions with stride 2 and padding equal to 1. Upsample layers can be also used that turn each pixel into n^2, where n is the amount of times the output will be bigger than the input.

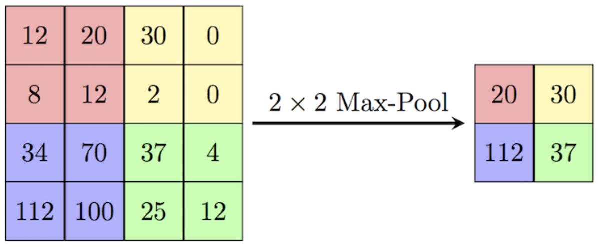

Figure 2.4: Simple example of a maxpool layer, taken from [9]

2.1.1.1.3 Fully Connected Layer

The fully connected layer is mostly used for classification in the final layers of the NN. It associates the feature map to the respective labels. It takes the 3D vector and outputs a single vector thus it is also known as flatten. Equation 2.3 describes the operation.

$$x_j^{l+1} = \delta(\sum_i (x_i^l \times w_{ji}^{l+1}) + b_j^{l+1}) \tag{2.3}$$

where w_{ji}^{l+1} are the weights associated with a specific input for each output.

2.1.1.1.4 Route & Shortcut Layer

The Shortcut layer or skip connection was first introduced in Resnet [10]. It allows to connect the previous layer to another to allow the flow of information across layers. The Route layer, used in Yolov3 [11], concatenates 2 layers in depth (channel) or skips the layer forward. This is used after the detection layer in Yolov3 to extract other features.

2.1.1.1.5 Dropout Layer

This type of layer was conceived to avoid overfitting [12] by dropping the neurons with probability below the threshold. In Figure 2.5, there is a graphical representation.

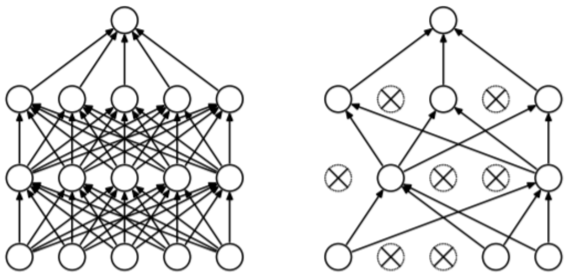

Figure 2.5: Dropout if applied to all layers, adapted from [12]

Activation Functions	Computation Equation
Sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$
Tanh	$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
Softmax	$f(x_i) = \frac{x_i}{\sum_j e^{x_j}}$
ReLU	$f(x) = \begin{matrix} x & if & x \geq 0 \\ 0 & if & x < 0 \end{matrix}$
LReLU	$f(x) = \begin{matrix} x & if & x > 0 \\ \alpha x & if & x \leq 0 \end{matrix}$
ELU	$f(x) = \begin{matrix} x & if & x > 0 \\ \alpha e^{x} - 1 & if & x \leq 0 \end{matrix}$

Table 2.1: Popular activation functions

2.1.1.1.6 Activation Functions

Activation Functions (AF) are functions used in each layer of a NN to compute the weighted sum of input and biases, which is used to give a value to a neuron. Non-linear AFs are used to transform linear inputs to non-linear outputs. While training Deep Neural Networks, vanishing and exploding gradients are common issues, in other words, after successive multiplications of the loss gradient, the values tend to 0 or infinity and thus the gradient disappears. AFs help mitigate this issue by keeping the gradient within specific limits. The most popular activation functions can be found in table 2.1.

2.1.2 Frameworks for Neural Networks

To run a Neural Network model there are several popular frameworks like Tensorflow, PyTorch, Caffe and Darknet. Their purpose is to offer abstraction to software developers that want to run these networks. They also offer programming for different platforms like nVidia GPUs by using the CUDA API.

2.1.2.1 Darknet

Darknet [13] is an open source neural network framework written in C and CUDA. It is used as the backbone for Yolov3 [11] and supports several different network configurations such as AlexNet and Resnet. It utilizes a network configuration file (.cfg) and a weights file (.weights) as input for inference.

Listing 2.1: cfg code for a Convolutional Layer used in Yolov3 [11]

```
[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky
```

In Listing 2.1, there is a snippet of the file featuring a convolution layer with 32 kernels of size 3x3. It has stride 1 and zero padding of 1, meaning the output size equals the input size. The input size can be calculated by analyzing the previous layers and the network parameters. The network parameters in Listing 2.2 includes data to be used for training while only the first three parameters are needed for inference.

Listing 2.2: cfg code for the network parameters

```
[net]
width=608
height=608
channels=3

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1
```

2.1.2.2 Caffe

Convolutional Architecture for Fast Feature Embedding (Caffe) [14] is also an open source framework written in C++ with a Python interface. Caffe exports a neural network by serializing it using the Google Protocol Buffers (ProtoBuf) serialization library. Each network has 2 prototxt files:

- deploy.prototxt- File that describes the structure of the network that can be deployed for inference.
- train_val.prototxt- File that includes structure for training. it includes the extra layers used to aid the training and validation process.

The Python interface helps generate these files. For inference only the deploy file matters. In Listing 2.3, there is a snippet of a deploy file.

Listing 2.3: prototxt file for the input data and the first convolution layer of AlexNet [7]

```
name: "AlexNet"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 10 dim: 3 dim: 227 dim: 227 } }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
  }
}
```

2.2 Deep Versat

Versat is a Coarse Grained Reconfigurable Array (CGRA) Architecture. CGRAs are in-between Field Programmable Gate Arrays (FPGA) and general purpose processors (GPP). The former is fully reconfigurable and the highest performance for a workload can be achieved as the Architecture is tailored to the workload. GPPs on the other hand, are nor reconfigurable and thus slower but are more generic and can process different workloads. While FPGAs have the granularity at the gate level, CGRAs have the granularity at the functional unit level. They are configurable at run-time and the datapath can be changed in-between runs.

In this chapter, the base Versat Architecture will be explained and then the Deep Versat Architecture and its improvements.

2.2.1 Versat Architecture

The Versat Architecture [15–18] is depicted in Figure 2.6. Its composed by the following modules: DMA,Controller,Program Memory,Control File Registry,Data-Engine and Configuration module. The Controller accesses the modules through the control bus. The code made in assembly or C is loaded into the program Memory (RAM) where the user can write to the configuration module for the versat runs. Between runs of the Data Engine, the Controller can start doing the next run configuration and calculations.

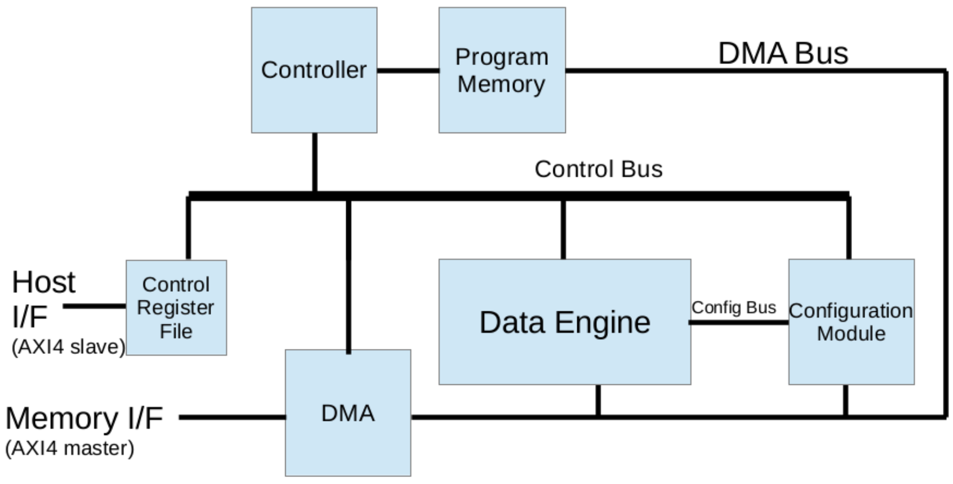

Figure 2.6: Versat Topology, taken from [16]

2.2.1.1 Data Engine

The Data Engine which is represented in Figure 2.7 carries out the computation needed on the data arrays. Its a 32 bit Architecture with up to 11 Functional Units (FU): Arithmetic and Logic Unit(ALU), stripped down ALU (ALU-Lite), Multiplier and Accumulator (MAC) and Barrel Shifter. Depending on the project and calculations, a new type of FU or the existing ones can be altered to support the algorithm. The DE has a full mesh topology, that means that each FU can be the output to another, which leads to a decrease in operating frequency.

Each Input of a Functional Unit has a Mux with 19 entries, 8 of which are from the memories (2 from each Mem out of 4 total units) and the rest from the Functional Units (11).

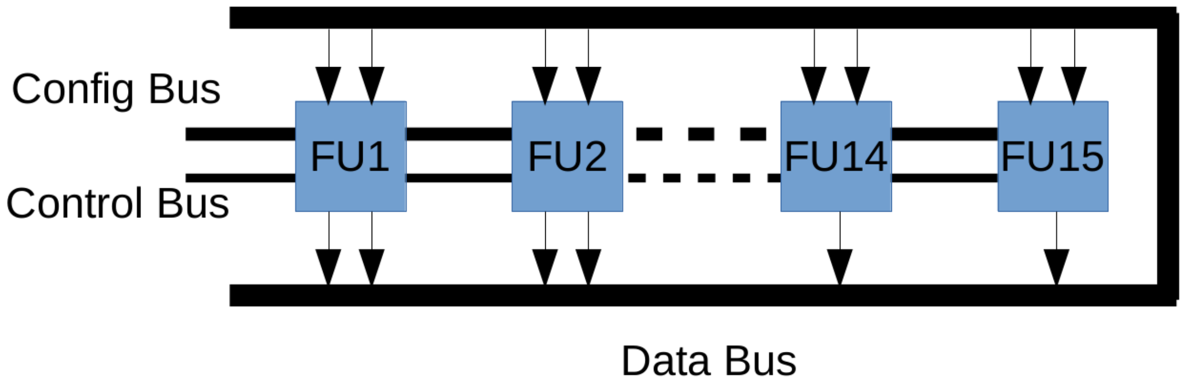

Figure 2.7: Versat Data Engine Topology, taken from [17]

The 4 Memories are dual port and for the input of both ports, there is an Address Generation Unit (AGU) that is able to reproduce two nested loops of memory indexes. The AGUs control which MEM data is the input of the FUs and where to store the results of the operation. Also, the AGUs support a delayed start to line up timings due to latencies.The memory module is represented in Fig 2.8.

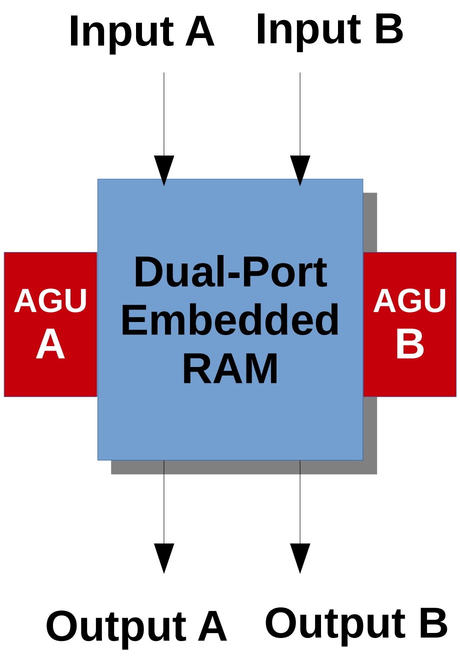

Figure 2.8: Versat Memory Unit with one AGU per port, taken from [19]

2.2.2 Configuration Module

Versat has several configuration spaces devised for each Functional Unit, with each space having multiple fields to define the operation of the Functional unit (e.g which op for the ALU). These are accessed before the run by the controller to define the datapath.

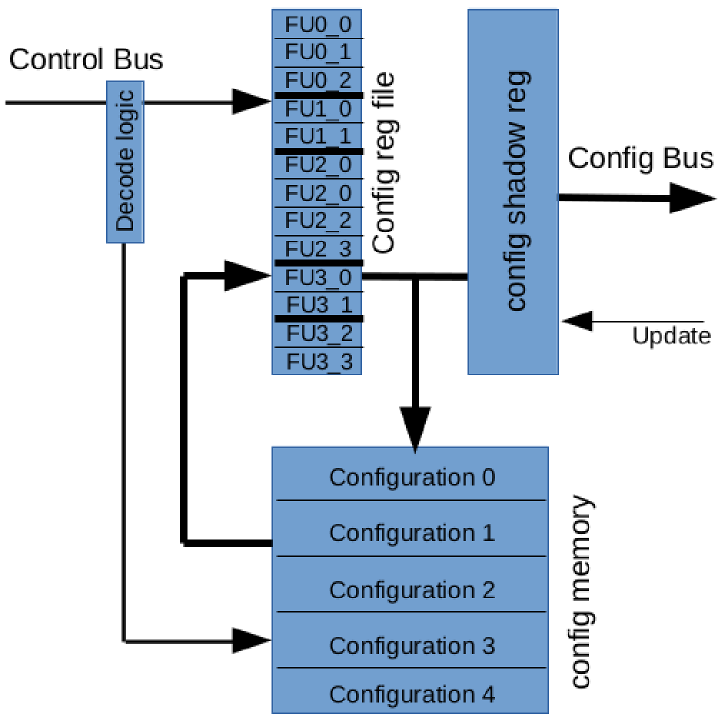

Figure 2.9: Configuration Module,taken from [16]

The Configuration Module (CM), depicted in figure 2.9, has three components:configuration memory, variable length configuration register file and configuration shadow register. The latter holds the current configuration so the controller can change the values of the configuration file in-between runs. The decode logic finds which component to write or read, if its the registers, it ignores read operations. Meanwhile, the configuration memory interprets both write and reads. When it receives a read, it writes into the register configuration data, when its a write, it stores the data instead.

2.2.3 Deep Versat Architecture

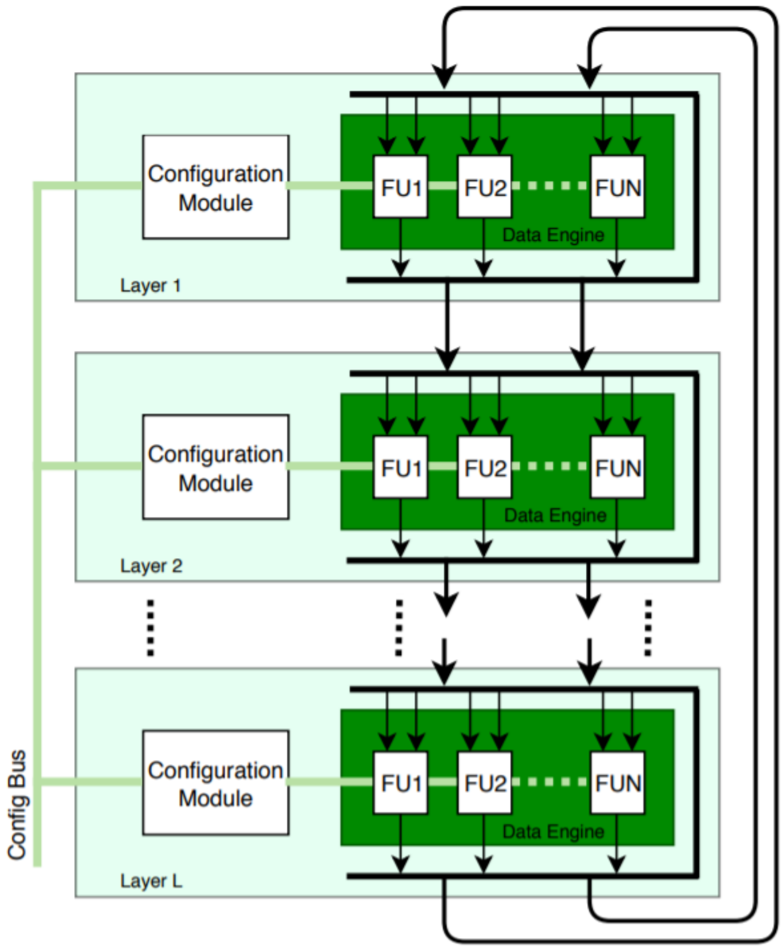

Figure 2.10: Deep Versat Architecture, taken from [1]

The Deep Versat Architecture [1] , in figure 2.10, decouples the Data Engine (DE) from all control and as such, it can be used with any CPU. It can be paired with hard cores in FPGA boards like the ZYNC board with its A9 ARM dual core CPUs or pair it with a soft core.

Its principle is to create the concept of a Versat Core: Configuration Module (CM) and its Functional Units (FU) connected with a control bus and a data bus. Instead of writing to a memory, there is the option to write for the next Versat Core to create more complex and more complete Datapaths, to avoid having to reconfigure the cores.

The number of Layers and FUs are reconfigurable pre-silicon with the only limitation that each layer is identical. To program Deep Versat, an API is generated from the Verilog .vh files.

2.2.3.1 Deep Versat System

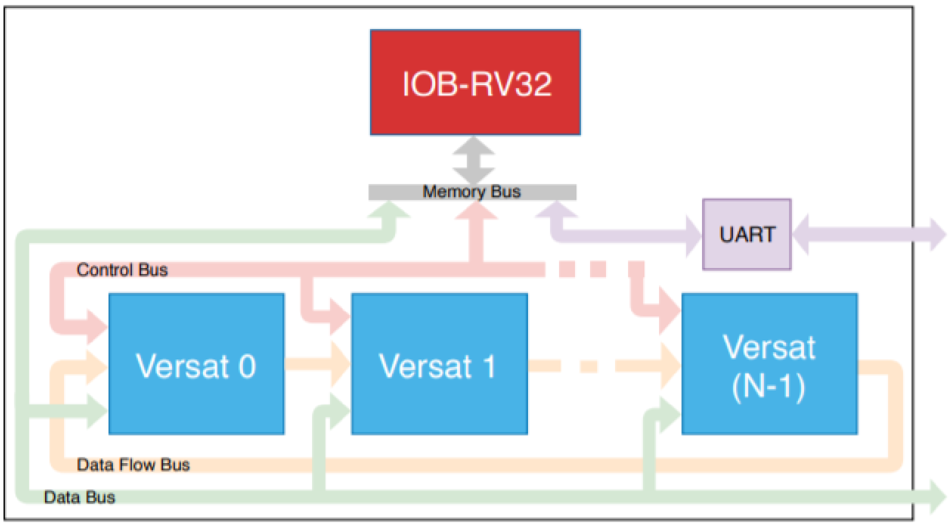

Figure 2.11: Deep Versat System using a RISC-V RV32IMC soft core, taken from [1]

To make a complete system, a new controller is needed with a more robust toolchain. In a recent dissertation [1], the IOB-RV32 processor was used which uses the RISC-V Instruction Set (ISA) with 32 bit Integer base alongside Multiplication and Division extension and Compact Instruction extension. The core is derived from the open source PicoRV32 CPU [2]. The IOB-RV32 uses its memory bus to access peripherals in which Deep Versat and the UART module are connected as such. The control bus is used to access the configuration modules of Deep Versat. The data bus is used to read and write large amount of data into Deep Versat. The data flow bus is reserved for inter Versat Core communication.

Peripheral	**Memory address**
UART module	12'h100xxxxx
Deep Versat control bus	8'h11xxxxxx
Deep Versat data bus	8'h12xxxxxx

Table 2.2: Deep Versat Memory Map

The memory map to address the peripherals, including deep versat, is in table 2.2. Each Versat has 15 bits of address while the CPU addresses the peripherals with 32 bits, with 8 of those occupied to chose the peripheral in question. That leaves 9 bits to address several Versat Cores which brings the theoretical maximum versat cores to 512. The IOB-RV32 is compatible with the GNU toolchain to offer better portability of code and alongside the C++ Versat API the difficulty to code for the System diminishes.

2.3 CNN Compiling in FPGAs

This chapter presents an overview of toolflows that map convolutional neural networks into FPGA using the frameworks presented in Section 2.1.2. Next, the concepts for mapping CNNs into CGRAs are introduced.

2.3.1 Toolflows for Mapping CNNs in FPGAs

Several software frameworks have been developed to accelerate development and execution of CNNs. The neural networks frameworks discussed in section 2.1.2 provide high level APIs together with high performance execution on multi-core CPUs, GPUs, Digital Signal Processors (DSPs) and Neural Processing Units (NPUs) [20]. FPGAs provide an alternative to these architectures as they provide high-performance while also being low-power. FPGAs can meet several requirements like throughput and latency in diversity of applications. Thus, several toolflows that map CNN descriptions into hardware in order to perform inference have been created. In table 2.3, a list of notable ones is presented.

Toolflow Name	Interface	Year
fpgaConvNet	Caffe & Torch	May 2016
DeepBurning	Caffe	June 2016
Angel-Eye	Caffe	July 2016
ALAMO	Caffe	August 2016
Haddoc2	Caffe	September 2016
DNNWeaver	Caffe	October 2016
Caffeine	Caffe	November 2016
AutoCodeGen	Proprietary Input Format	December 2016
Finn	Theano	February 2017
FP-DNN	Tensorflow	May 2017
Snowflake	Torch	May 2017
SysArrayAccel	C	June 2017
FFTCodeGen	Proprietary Input Format	December 2017

Table 2.3: CNN to FPGA Toolflows, adapted from [21]

2.3.1.1 Supported Neural Network Models

These toolflows support the most common layers in CNNs, which are discussed in chapter **??**. The acceleration target changes depending on the toolflow. For example, the fpgaConvNet [22] toolflow focuses more on feature extraction while offering non accelerated support for fully connected layers.

2.3.1.2 Architecture & Portability

As shown in figure 2.12, the fpgaConvNet architecture consists of a Front-End Parser that reads a (ConvNet) description of the network and a description of the target platform and produces, on the one hand a Directed Acyclic Graph (DAG), which is then converted to a Synchronous Data Flow (SDF) hardware model, and on the other hand, a model of the target platform from which resource constraints are derived. The hardware model thus obtained goes into an Optimiser procedure, which produces a hardware mapping. Using hardware and software templates, a Code Generator procedure, generates both the High Level Synthesis (HLS) input files and the software binaries that will run on the control CPU embedded in the FPGA. The HLS files go into the Xilinx (FPGA manufacturer) tools so that the configuration bitstream of the FPGA is produced.

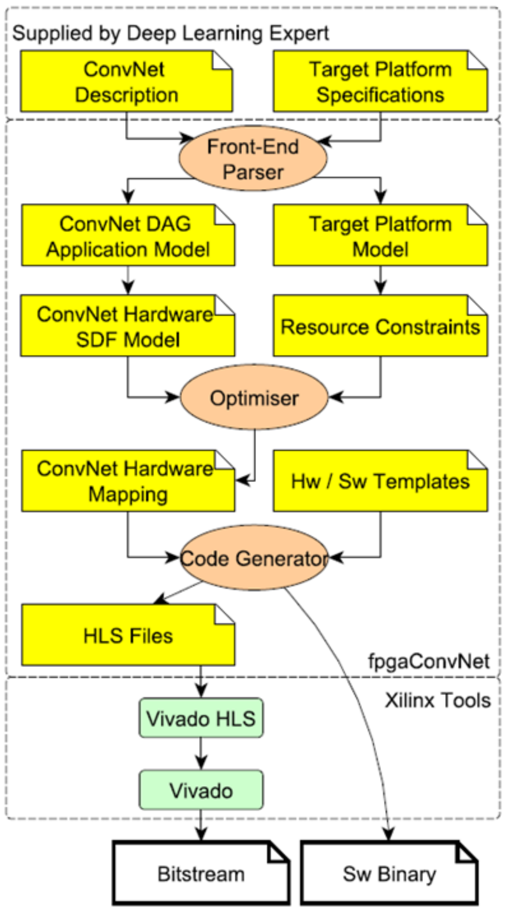

Figure 2.12: fpgaConvNet Architecture. Taken from [22]

Chapter 3

Darknet Lite

As mentioned in Section **??**, the Deep Versat system includes a RISC-V CPU to take out generic code and to write the configuration runs into Versat's memories. This means the first step into implemetning software that can run any convolutional neural network on this system, it must first run on the CPU then we off load Fixed Functions to Versat such as the convolutional layers, maxpool etc.

3.1 Porting Darknet to an embedded CPU

As mentioned in Section 2.1.2 is a framework for Neural Networks on C++ that uses dynamic memory and GPU acceleration option to get faster outputs. Also the use of floats is also prohibited in the embedded code as the RISC-V CPU only supports the extentions IM. I for Integer and M for multiplication. It also has a lot of features that are not needed in this work, such as training the CNN. By stripping the features of darknet we get a much simpler code framework apropriately named darknet lite.

In the following figure, the data strucuture for a layer is showed. A CNN on darknet lite is just an array of layers in which each has an input, output and layer parameters. Usually the input is a past layer output or the image input.

Listing 3.1: Layer Struct Yolov3 [11]

```
struct layer{
             //Generic
                LAYER_TYPE type; //identifies layer's type
                ACTIVATION activation; //identifies layer's activation function
                void (*forward) (struct layer, struct network); //associated with forward
                    method of each type of layer
                int groups;
             // Convolutional
                int batch_normalize; //indicates layer output must be normalized before
                    applying activation function
                int batch; //always 1
```

```
        int inputs; //size of layer input
        int outputs; //size of layer output
        int h,w,c; //input dimensions
        int out_h, out_w, out_c; //output dimensions
        int n; //number of filters
        int size; //size of filter
        int stride; //indicates how many positions kernel moves
        int pad; //indicates size of padding sorrounding image

    //Shortcut
        int index; //used in shortcut layer

        int classes; //used in yolo layer
        int *mask; //used in yolo layer
        int total; //used in yolo layer
        int  * input_layers; //used in route layer
        int  * input_sizes; //used in route layer
        fixed_t * biases; //used for convolutional and yolo layers
        fixed_t * scales; //used for convolutional layers with batch_normalize
        fixed_t * weights; //convolutional layer weights
        fixed_t * output; //layer output /result
        fixed_t * rolling_mean; //used for normalize_cpu
        fixed_t * rolling_variance; //used for normalize_cpu
        size_t workspace_size; //indicates max output size among all layers



        //Generic Var
        fixed_t f1; // float->fixed 32 bit
    };
```

By Parsing the .cfg file, a configuration file is written in C with the layer array and static position of the data for each layer. Each Layer has it's definition in C to be ran by the embedded CPU but for the sake of this project, several layers can be replaced by Functions that utilize Versat, the same way that the original Darknet framework had it's functions written for CPU or GPU usage.

In the following figure is an example of a CPU layer that computes the convolutional layer while using Fixed Point Logic.

Listing 3.2: Convolutional Layer using only CPU and fixed memory

```
void forward_convolutional_layer(layer l, network net) {


   int m = l.n; //number of filters
   int k = l.size*l.size*l.c; //filter dimensions * number of colours
```

```
    int n = l.out_w*l.out_h; //output dimension

   fixed_t *a = l.weights; //weight base address
   fixed_t *b = net.workspace; //max network's layer size
   fixed_t *c = l.output; //layer output
   fixed_t *im = net.input; //layer input


   //Unroll image
   if (l.size == 1) b = im;
   else im2col_cpu(im, l.c, l.h, l.w, l.size, l.stride, l.pad, b);
   //Perform convolution
   gemm(0, 0, m, n, k, POINT, a, k, b, n, POINT, c, n);



   //Normalize, scale and add bias
    if(l.batch_normalize) forward_batchnorm_layer(l, net);
    else add_bias(l.output, l.biases, l.batch, l.n, l.out_h*l.out_w);


   //Apply activation method
    activate_array(l.output, l.outputs*l.batch, l.activation);
    //printf("max=%f,min=%f\n",fixed_to_float(max),fixed_to_float(min));



}
```

3.2 Conversion of Caffe to CFG

3.3 Parsing CFG Files into the program

Chapter 4

Deep Versat Software Simulator

The need for a software simulator comes from the complexity of the configurations being written into Versat and the hardware simulation faults of taking too much time and hard to debug.

The goal is to emulate what the hardware is doing much more efficienctly than a simple Hardware simulation as the time of development for hardware is much higher than simple software. The Simulator executes clock iteration per iteration getting the same results in each clock as the hardware. As Versat is a CGRA, different functional unit configurations are easy to accomplish in the simulator and the time to get results on performance for a specific program is a lot faster.

In this chapter, we will explore the software arquitecture, object relation and explain in detail each method used to emulate Versat clock by clock.

4.1 Architecture and Object Relation

The Simualtor is made up by the Parent Class called Versat, in which will be simulated itself, as each Versat instance is independent from each other, the simulations are also independent. The Versat is made up of 2 CStage Arrays, one is the ”live” while the other is the shadow registers, where the configurations are held before the simulator is ran. Each Stage is made up of it’s Functional Units, in which each one is connected to the Databus. As it happens in the hardware, functional units can access the databus which has the output of the current stage and previous one.

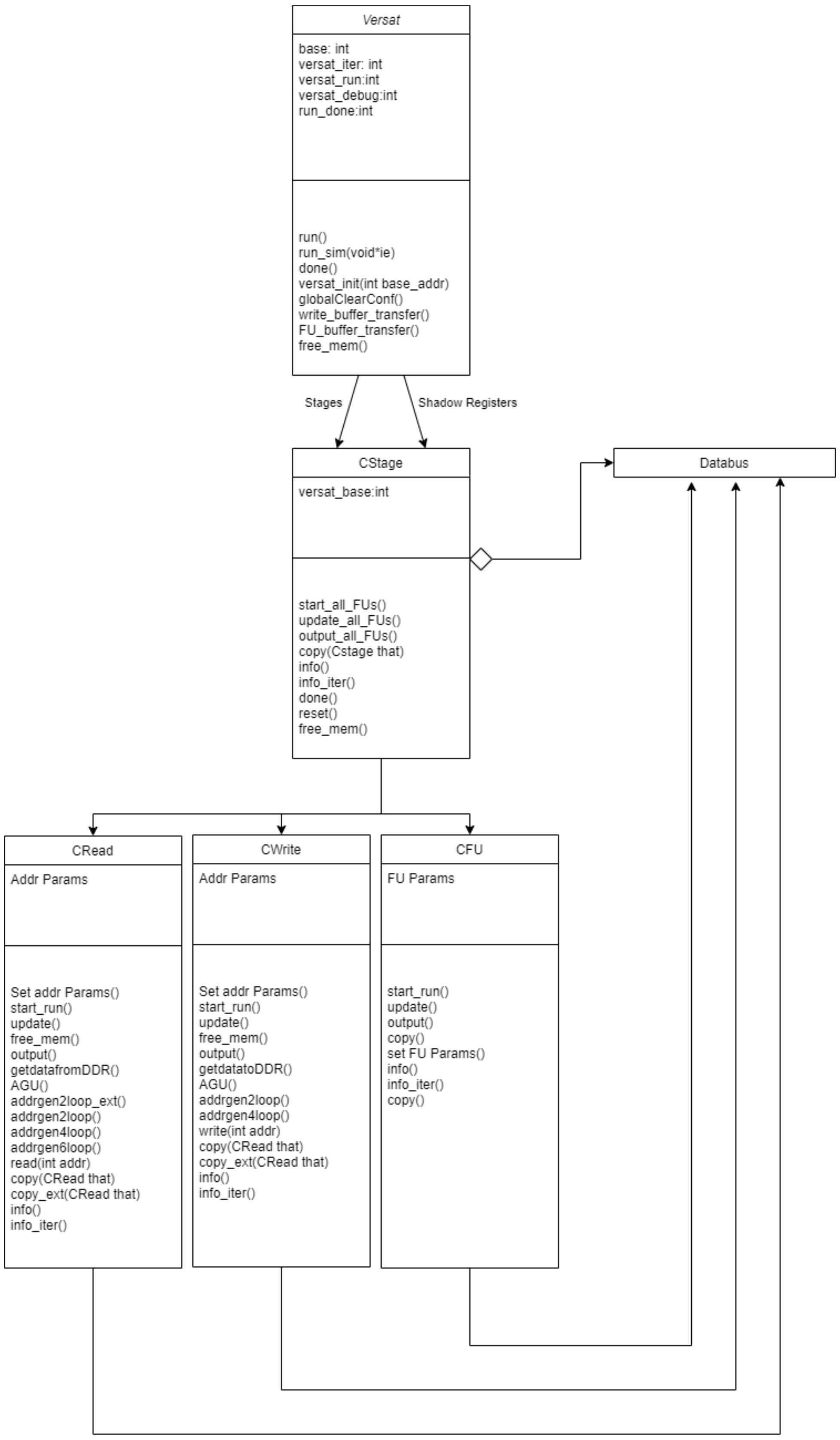

Figure 4.1: Class Structure for the Versat Simulator

4.1.1 Functional Units

The following table contains the functional units present in the simulator and is represented by ”CFU” in figure 4.1. VI and VO represent CRead and CWrite classes respectively.

Functional Unit	**Porpuse**
Read (VI) Mem Unit	Reads from DDR and sends Data to databus
Write (VO) Mem Unit	Reads from databus and sends Data to DDR
MulAdd (MAC)	Multiplication and Accumulate
Mul	Multiplication
Alu	Standard algorithmic and logic unit
AluLite	Stripped down algorithmic and logic unit
Barrel Shifter (BS)	Shifts to the right (division by 2) or to the left (multiplication by 2)
Memory (Mem)	Sends/Receives data to/from the pipeline. Data is inserted through CPU communication

Table 4.1: Versat Simulator Functional Units

To add a new FU, it’s as easy as to create a new class that will be used by CStage with a run(),update(),output(),copy() method. Of course, if it has variables needed to be defined by the program, set param functions are also needed. Using the simulator, hardware development and program development can be parallelized to output a new program with more optimized performance.

In the next section, these methods will be explained in detail and their importance to the simulator.

4.2 Simulation

After the program that is running on the CPU finishes writing the configurations, it will call the run method of Versat. On figure 4.2, a sequence diagram is presented with the rundown of a typical program that uses Versat Simulator.

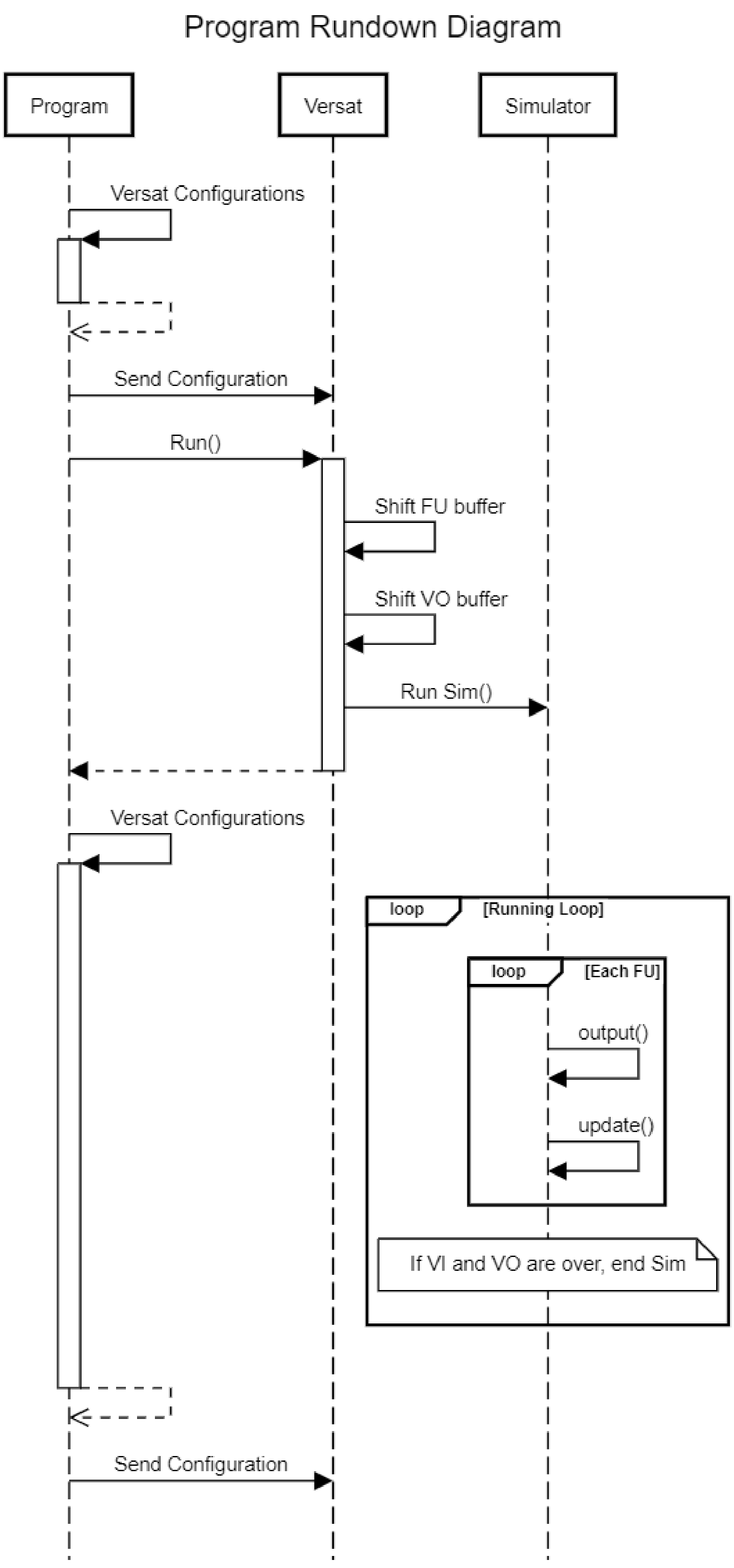

Figure 4.2: Sequence Diagram of a Program using Versat Simulator

4.2.1 Run() Function

In the software API for Embedded Versat, the run function would write to a shadow register, which we can call it ”start” changing the value from 0 to 1. Similarly, another register would change value to 0, which we can call ”done”. While this last register isn’t turned to 1, Versat hasn’t finished running with the previous configurations so all that can be done is to write configurations for future runs.

In the simulator, it works in a similar way to preserve compability as the goal is to have the same programs run on software simulator and on the FPGA.

Listing 4.1: The Run function code

```
void CVersat::run()
{
    //MEMSET(base, (RUN_DONE), 1);
    run_done = 0;
    versat_iter = 0;

//update shadow register with current configuration
#if nVO > 0
    write_buffer_transfer();
#endif
#if nVI > 0
    FU_buffer_transfer();
#else
    int i = 0;

    for (i = 0; i < nSTAGE; i++)
    {
        stage[i].reset();
        shadow_reg[i].copy(stage[i]);
    }
#endif


    pthread_create(&t, NULL, run_simulator,(void*)this);
}
```

As we can see in the previous listing, we reset state variables of the simulator, then shift the VO and FU shadow registers. This is done to simulate the pipeline delay in the FPGA. Because the data needs to come and go to main memory (DDR), 1 run cycle is used just for fetching data and writting data. Using a small example: If a developer writes a configuration to do a 5x5 matrix multiplication, Versat will have to run 3 times. Once to fetch data from memory, the second for the actual use of Versat and the final one to get data onto memory.

In the simulator, this is done using the same class instances and copying the configuration values. On the hardware it's several flip-flop registers in a row. However, all these 3 stages can happen at once if you run multiple configurations in one program, e.g: running a CNN through Versat, it will have at least 1 run per layer. So, if it has 5 layers, Versat will have to run 5+2 times, the last 2 times are done to flush the Versat of any data.

After the shift, a new thread is created to run the simulator in parallel with the configurations, having the same behaviour as the hardware.

4.2.2 Start() Method

At the beginning of the configuration run, the method "start run" of all FUs and memories is started. In this function, several functional units will have it's state variables reset such as VI,VO and MAC FU.

4.2.3 Databus

The databus on Versat is a simple array that holds all the outputs of the functional units. The data type (versat_t) of the array depends on the width of Versat, which is part of the configuration file. Using higher width, e.g: 64 bits, is useful for same instruction, multiple data (SIMD) applications but requires the functional units to be adapted. For the porpuse of this thesis, 16 bits and 32 bits are used depending on the neural network and how it is optimized.

When the versat is instanced in the program, the functional units constructor will point to the correct position of the databus as it's referenced in the following figure.

As mentioned in figure 2.10 from chapter 2, section 2.2, each functional unit will be able to access the output from the functional units of the current stage and previous. Software wise, each stage will be pointing to a part of the databus.

4.2.4 Update() and Output() Method

The update method's goal is to update the functional unit's value on the databus. Each functional unit has a pipeline delay to output or has a run delay configured, like the memories or MAC.

Meanwhile, the output method's goal is to, based on the inputs from the databus, calculate the result from the functional Unit.

For a compute functional unit such as the MAC or the ALU, this means reading from the databus for operands A and B and performing the selected operation. For the read memory (VI), it will output an address on the mem and performs a read operation. For the write memory, it will output an address and performs a write operation.

In the listing 4.3, the code of the Mul functional unit is used as an example.

Listing 4.2: Update and Output method of Mul

```
void CMul::update()
{
    int i = 0;

    //update databus
    databus[sMUL[mul_base]] = output_buff[MUL_LAT - 1];
    //special case for stage 0
    if (versat_base == 0)
    {
        //2nd copy at the end of global databus
        global_databus[nSTAGE * (1 << (N_W - 1)) + sMUL[mul_base]] = output_buff[MUL_LAT - 1];
    }

    //trickle down all outputs in buffer
    for (i = 1; i < MUL_LAT; i++)
    {
        output_buff[i] = output_buff[i - 1];
    }
    //insert new output
    output_buff[0] = out;
}

versat_t CMul::output()
{
    //select inputs
    opa = databus[sela];
    opb = databus[selb];

    mul_t result_mult = opa * opb;
    if (fns == MUL_HI)
    {
        result_mult = result_mult << 1;
        out = (versat_t)(result_mult >> (sizeof(versat_t) * 8));
    }
    else if (fns == MUL_DIV2_HI)
    {
        out = (versat_t)(result_mult >> (sizeof(versat_t) * 8));
    }
    else // MUL_LO
    {
        out = (versat_t)result_mult;
    }


    return out;
}
```

4.2.5 Copy() and Info() Method

Finally, the last functions of the simulator, copy() main porpuse is to copy the configuration parameters from instance to another, used mostly in the beginning of the run to simulate the shadow registers. Meanwhile the Info method is a State printing function that outputs a string with the full data of the current iteration, this way, you can check iteration by iteration the progress of the simulation, just like in hardware. At this moment, if debugging is activated, each clock iteration output and state of versat will be in a file.

Listing 4.3: Info output for the MAC functional unit

```
mul_add[0]
OpA=    -7
OpB=    -3
SelA=    1
SelB=    0
Addr=    3
Finished=    0
Out=    61
OUTPUT_BUFFER (LATENCY SIM)
Output[0]=61
Output[1]=40
Output[2]=60
Output[3]=45
```

Chapter 5

Versat API 2.0

The Versat API, developed in a previous thesis [1], has the ability to conceal the calls to the hardware to avoid changing the program when the hardware changes.

In this chapter, the new functions that are part of the Versat API will be discussed. The goal is to make development for Versat just like writting normal code and to be easy to port code to it just like CUDA has done the same to run SIMD code on Nvidia GPUs.

Listing 5.1: Sample Versat API implementation for the Hardware for Mem functional unit

```
class CMemPort
{
public:
  int versat_base, mem_base, data_base;

  //Default constructor
  CMemPort()
  {
  }

  //Constructor with an associated base
  CMemPort(int versat_base, int i, int offset)
  {
    this->versat_base = versat_base;
    this->mem_base = CONF_BASE + CONF_MEM0A + (2 * i + offset) * MEMP_CONF_OFFSET;
    this->data_base = (i << MEM_ADDR_W);
  }

  //Methods to set config parameters
  void setIter(int iter)
  {
    MEMSET(versat_base, (this->mem_base + MEMP_CONF_ITER), iter);
  }
  void setPer(int per)
```

On figure 5.1, a graphic representation of the new API is presented. It has 4 apparent layers (5 if you count the hardware):

1. Complex Matematical API that is automatically optimized for the Versat Setup you chose. No dev work required
2. Read/Write using VI and VO for simpler setup of the data. Also includes easier FU functions to setup workloads.
3. Read/Write configurations for inside Versat Data (Int) or DDR to/from VI/VO (Ext).
4. Versat API 1.0 where each configuration variable needs to be set up individually
5. No API. Hardware registers where the values are used inside Versat.

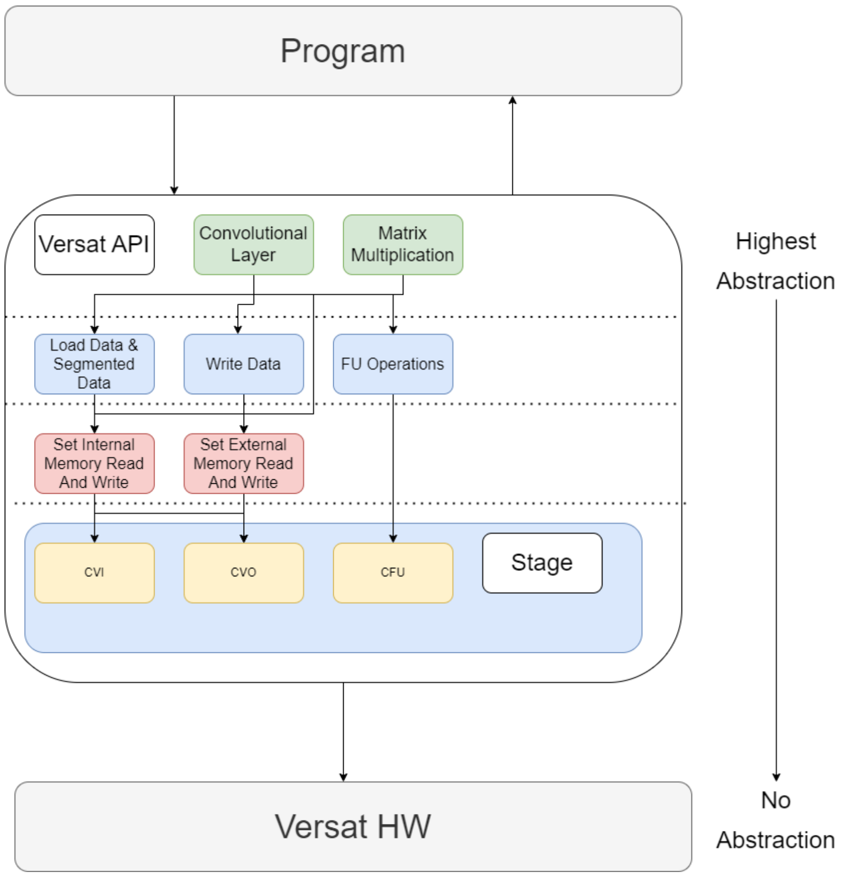

Figure 5.1: Graphic representation of the new Versat API and it's connections

5.1 Memory Operations API

When utilizing the VI instead of a MEM, the data transfer happens between the functional unit and a Direct memory access while on the mem, the CPU writes directly to Versat, wasting CPU cycles. For the API, this means going from a read method that is straightforward to more configuration methods to set up the read operation from DDR The same happens to Write operations. To address this, 7 functions were created in 2 levels of abstraction.

load_data(),load_segmented_data(),write_data() that use a lower level functions: set_IntMem_Write(),set_ExtMem_Writ
and set_ExtMem_Read().

The first three functions use the lower level

5.1.1

5.2 Software Layers

Due 18/10

5.3 Generic Convolution API

due 18/10

5.3.1 Hardware Configurations

due 18/10

5.3.2 Abstracting Versat Configuration

due 18/10

5.3.3 Convolution Scenarios

due 18/10

Chapter 6

Conclusions

Insert your chapter material here...

6.1 Achievements

The major achievements of the present work...

6.2 Future Work

A few ideas for future work...

Bibliography

[1] V. J. B. Mário. Deep versat: A deep coarse grain reconfigurable array. Master's thesis, Instituto Superior Técnico, November 2019.

[2] Picorv32- a size-optimized risc-v cpu. URL `https://github.com/cliffordwolf/picorv32`.

[3] G. Piccinini. The first computational theory of mind and brain: A close look at mcculloch and pitts's "logical calculus of ideas immanent in nervous activity". *Synthese*, 141, 08 2004. doi: 10.1023/B:SYNT.0000043018.52445.3e.

[4] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015. doi: 10.1038/nature14539.

[5] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(05)80131-5. URL `http://www.sciencedirect.com/science/article/pii/S0893608005801315`.

[6] mnist database of hand-written digits. URL `http://yann.lecun.com/exdb/mnist/`.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.

[8] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima. A cgra-based approach for accelerating convolutional neural networks. pages 73–80, 09 2015. doi: 10.1109/MCSoC.2015.41.

[9] Max-pooling / pooling. URL `https://computersciencewiki.org/index.php/Max-pooling_/_Pooling`.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[11] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[13] J. Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014. URL `http://arxiv.org/abs/1408.5093`.

[15] R. Santiago, J. D. Lopes, and J. T. de Sousa. Compiler for the versat reconfigurable architecture. REC 2017, 2017.

[16] J. D. Lopes, R. Santiago, and J. T. de Sousa. Versat, a runtime partially reconfigurable coarse-grain reconfigurable array using a programmable controller. Jornadas Sarteco, 2016.

[17] J. D. Lopes and J. T. de Sousa. Fast fourier transform on the versat cgra. Jornadas Sarteco, 09 2017.

[18] J. D. Lopes and J. T. de Sousa. Versat, a minimal coarse-grain reconfigurable array. In D. I., C. R., B. J., and M. O., editors, *High Performance Computing for Computational Science – VECPAR 2016*, pages 174–187. Springer, 2016. doi:10.1007/978-3-319-61982-8_17.

[19] J. D. Lopes. Versat, a compile-friendly reconfigurable processor – architecture. Master's thesis, Instituto Superior Técnico, November 2017.

[20] A. Ignatov, R. Timofte, P. Szczepaniak, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool. Ai benchmark: Running deep neural networks on android smartphones, 10 2018.

[21] S. I. Venieris, A. Kouris, and C.-S. Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions, 2018.

[22] S. I. Venieris and C.-S. Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. Institute of Electrical and Electronics Engineers (IEEE), May 2016. doi: 10.1109/FCCM.2016.22. URL `http://dx.doi.org/10.1109/FCCM.2016.22`.