TÉCNICO LISBOA

Deep Neural Networks 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

Chairperson: Prof. Teresa Maria Canavarro Menéres Mendes de Almeida
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Mario Pereira Véstias

May 2023

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 and my firstborn 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 through my Bachelor's and Master's and has finally pushed me to finish this document.

Resumo

O foco desta tese concentra-se na aceleração de Redes Neuronais Profundas (DNN) com os recursos da matriz reconfigurável de grão grosso (CGRA) DeepVersat. O objetivo principal é desenvolver um compilador que converta as descrições DNN em código executável otimizado para o sistema CPU/DeepVersat. Para conseguir isso, uma estrutura de rede neuronais, Darknet, é estendida, adaptada e simpleficada para compilar arquivos de descrição DNN em código que integra-se com o sistema, utilizando a interface de software (API) do Versat. A API do Versat foi expandida para conseguir a aceleração de camadas de computação intesiva, com alocação dinâmica de recursos para melhorar o desempenho. O simulador em software também foi desenvolvido para facilitar a otimização arquitetônica e reduzir o tempo de desenvolvimento para implementações baseadas no DeepVersat. A utilidade do Darknet Lite na compilação de DNNs no código Versat e a eficácia da nova API em várias condigurações de hardware são demonstradas por vários ficheiros de teste, estabelecendo uma prova de conceito para a abordagem proposta.

Palavras-chave: Matrizes Reconfiguráveis de Grão Grosso, Versat, Darknet, Redes Neuronais Convolucionais, Redes Neuronais Profundas, Simulador, Sistemas Heterógenos, Sistemas Embebidos

x

Abstract

This thesis focuses on accelerating Deep Neural Networks (DNN) with the capabilities of the DeepVersat Coarse-Grained Reconfigurable Array (CGRA). The primary objective is to develop a compilation approach that converts DNN descriptions into executable code optimized for CPU/DeepVersat system. To achieve this, a neural network framework, Darknet, is extended, adapted, and streamlined to compile DNN description files into code that integrates with the system, utilizing the Versat Application Programming Interface (API). The Versat API is expanded to enable acceleration of compute-intensive layers, with dynamic resource allocation for improved performance. A software simulator is also developed to facilitate architectural optimization and reduce development time for DeepVersat-based implementations. The usefulness of Darknet Lite in compiling DNNs into Versat code and the effectiveness of the new API on various hardware configurations are demonstrated through multiple test files, establishing a proof of concept for the proposed approach.

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

NPU Neural Processing Unit

CGRA Coarse-Grain Reconfigurable Array

CM Configuration Module

CNN Convolutional Neural Network

DNN Deep Neural Network

CPU Central Processing Unit

DE Data Engine

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

FU Functional Unit

ISA Instruction Set Architecture

DAG Directed Acyclic Graph

SDF Synchronous Data Flow

HLS High Level Synthesis

NN Neural Network

FP32 Floating Point 32 bit

MLP Multilayer Perceptrons

GPP General Purpose Processor

RAM Random Access Memory

MAC Multiplier and Accumulator

IP Intellectual Property

SIMD Single Instruction Multiple Data

VI Versat Input Memory

VO Versat Output Memory

Chapter 1

Introduction

In this thesis, the problem of accelerating the execution of Deep Neural Networks (DNNs) using Coarse-Grained Reconfigurable Arrays (CGRAs) is studied. The emphasis is on compiling a DNN description into C-language code that runs on CPU/CGRA system, and simulating the execution using a software simulation model. The DeepVersat Architecture [1] CGRA is used as an implementation tool in this work.

1.1 Motivation

Neural Networks have been an object of study since the 1940s 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 the 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 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 fewer configuration bits. If reconfiguration is done at runtime, CGRAs add temporal scalability to the spacial scalability that characterizes FPGAs. Moreover, partial reconfiguration is much easier to do in CGRAs compared to FPGAs which further speeds up reconfiguration time. Another advantage of CGRAs are 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 Array (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 the time to market of its users. Currently, there are equivalent tools for CPUs and GPUs and even for FPGAs.

1.2 Objetive

The main objective of this thesis is to take an established Neural Network Framework, in this case Darknet[2] and accelerate the computational intensive workloads that will run on the DeepVersat CGRA. A tool will transform a prototype machine learning model file created for Caffe into CFG files which are read by Darknet, so if a user has a DNN in Caffe, it can be used by the system. Afterward, the CFG file can be parsed by the tool to create the layer and data structures needed for Darknet.

The Versat CGRA is the DNN accelerator to improve the performance of the DNNs in embedded hardware. This work presents a software simulator for Versat so the development can be simultaneous and to write the configurations of said hardware. Another objective is to increase the versatility of the Versat API and offer new functions to simplify the development of new software. One of these functions is a generic convolution for Versat which can, independently of the hardware configuration, configure the convolution to have the highest performance possible on the available functional units while being dynamic and to avoid developer work to adapt to new convolutions.

1.3 Thesis Outline

The document has the following chapters:

- Chapter 2 introduces the background needed to understand the work presented in other chapters relating to neural networks and the Versat CGRA.

- Chapter 3 describes the Darknet framework and its embedded implementation, the tool to transform Caffe to CFG and to transform CFG to C++ code with the layers and data structures needed

- Chapter 4 talks about the DeepVersat Simulator and how the simulator structure and architecture is designed and implemented.

- Chapter 5 explains the new functions that the Versat API has that are used for development

- Chapter 6 presents the results of the work explained in the previous chapters as well and the expected performance that the Versat CGRA has with several convolutions using the simulator.

- Chapter 7 is the final remarks of this thesis, explains the shortcomings and what's missing from this thesis and possible future work.

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 an 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.
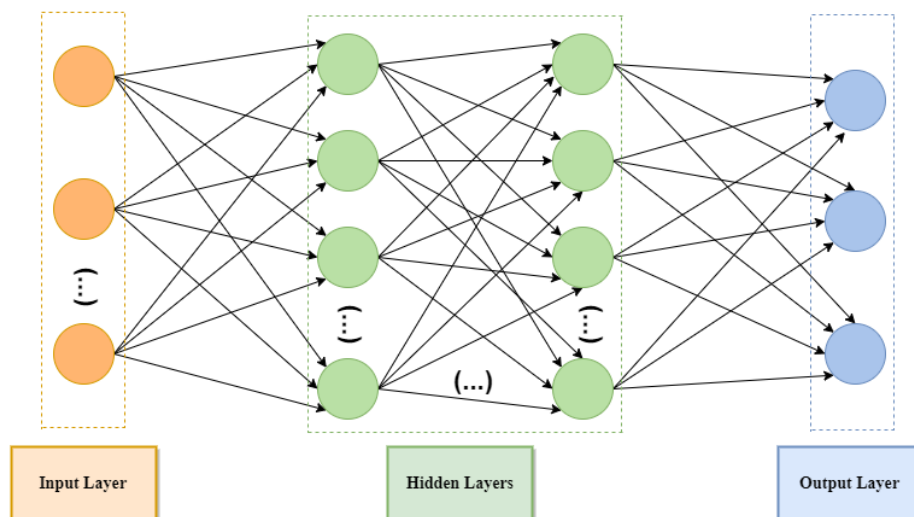

Figure 2.1: Deep Neural Network Structure

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 to improve 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 an 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 computational complexity grows quadratically with the number of neurons 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 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.
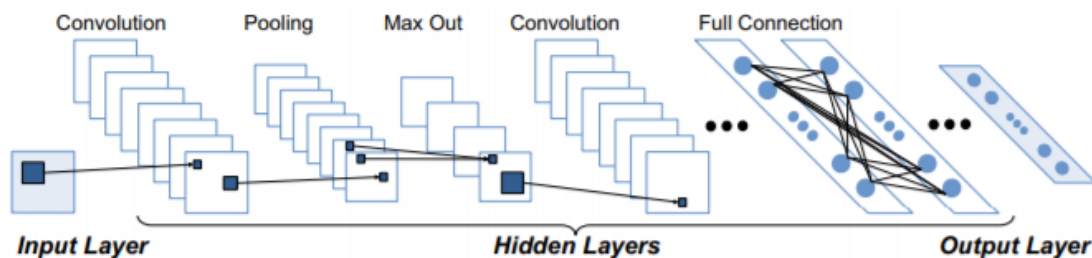
Figure 2.2: CNN architecture example, taken from [8]

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. CNNs take input images with three 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 the 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.
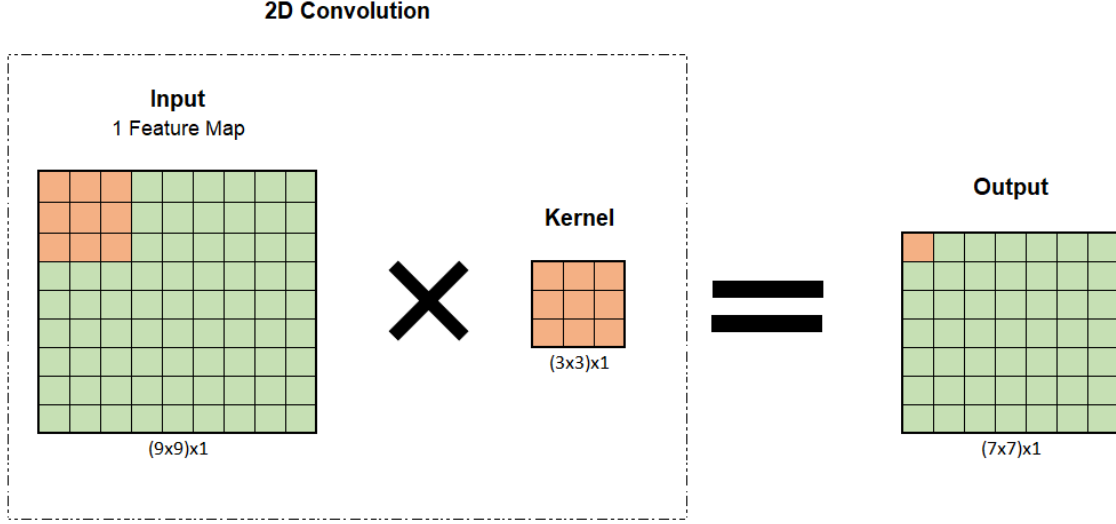
2D Convolution

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

The output's dimensions depend on convolution parameters such as zero-padding and stride. The former means to add zeros around the edges of the input matrix. The latter implies the step used for the convolution. If the value is, e.g., 2, it will skip a pixel each iteration. 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 is suggested in its name groups $n * n$ points and outputs the pixel with the 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 two and padding equal to 1. Upsample layers can also be utilized that turn each pixel into n^2, where n is the number of times the output will be bigger than the input.
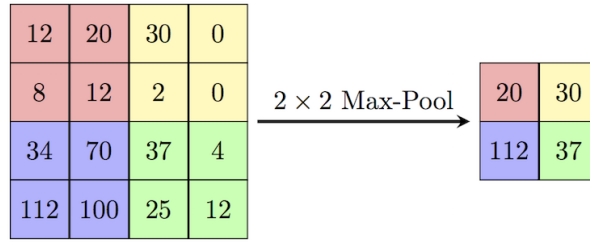
Figure 2.4: Simple example of a max pool layer, taken from [9]

2.1.1.1.3   Fully Connected Layer

The fully connected layer is mainly used for classification in the final layers of the NN. It associates the feature map with 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 connecting of the previous layer to another to allow the flow of information across layers. The Route layer, used in Yolov3 [11], concatenates two 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 a probability below the threshold. In Figure 2.5, there is a graphical representation.
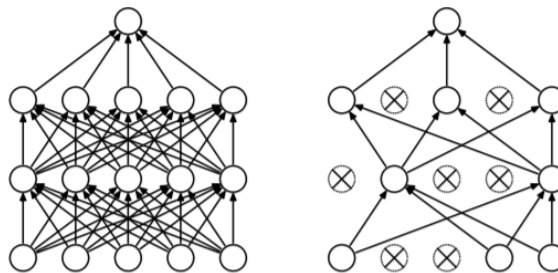
Figure 2.5: Dropout if applied to all layers, adapted from [12]

8

Activation Functions	Computation Equation
Sigmoid	$f(x) = \dfrac{1}{1 + e^{-x}}$
Tanh	$f(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
Softmax	$f(x_i) = \dfrac{x_i}{\sum_j e^{x_j}}$
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 into 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 zero 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. They aim to offer abstraction to software developers who want to run these networks. They also provide programming for platforms like Nvidia GPUs using the CUDA API.

2.1.2.1 Darknet

Darknet [2] is an open-source neural network framework written in C and CUDA. It is the backbone for Yolov3 [11] and supports several 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 one 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 include data 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) [13] 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 two prototxt files:

- deploy.prototxt- File that describes the network structure 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 deployed 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 DeepVersat

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 not reconfigurable and thus slower but are more generic and can process different workloads. While FPGAs have granularity at the gate level, CGRAs have 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 DeepVersat Architecture and its improvements.

2.2.1 Versat Architecture

The Versat Architecture [14–17] is depicted in Figure 2.6. It is composed of 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. Then, between runs of the Data Engine, the controller can start doing the following run configuration and calculations.

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. It is 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, which 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, eight of which are from the memories (2 from each Mem out of four total units) and the rest from the Functional Units (11).

11
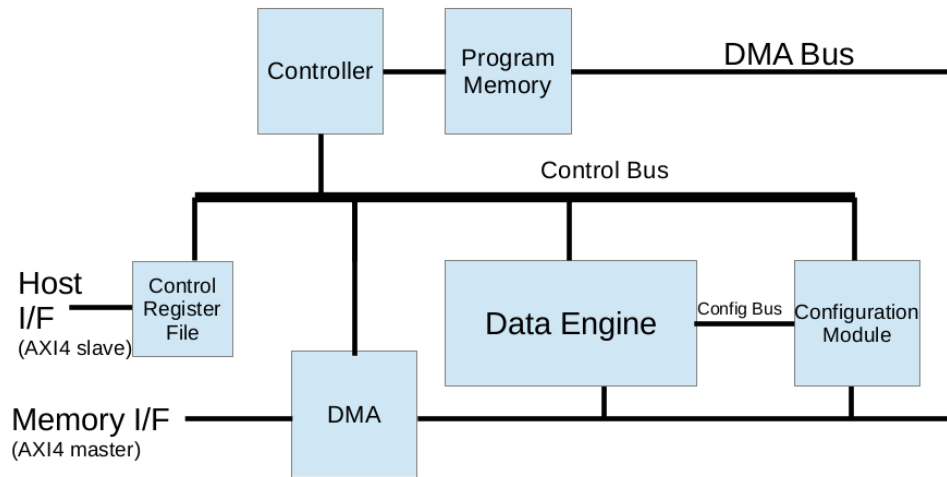
Figure 2.6: Versat Topology, taken from [15]
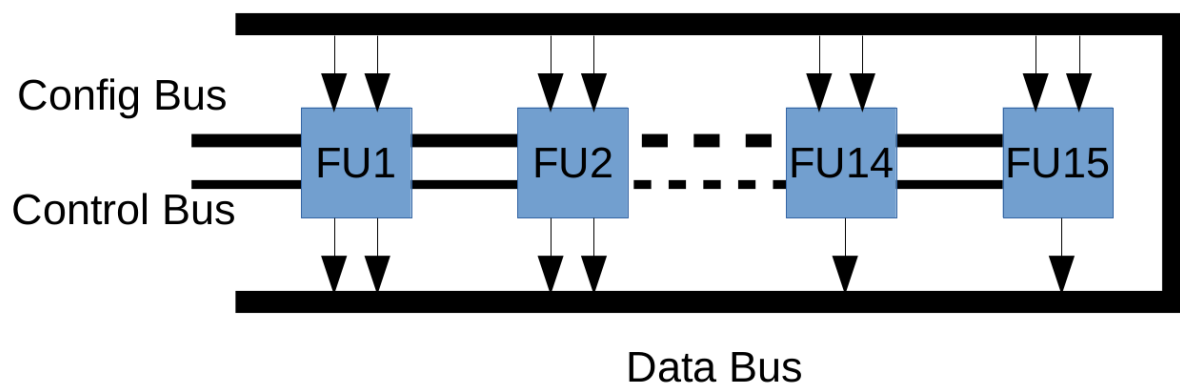
Figure 2.7: Versat Data Engine Topology, taken from [16]

The four 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 operation results. Also, the AGUs support delayed start to line up timings due to latencies. The memory module is represented in Fig 2.8.
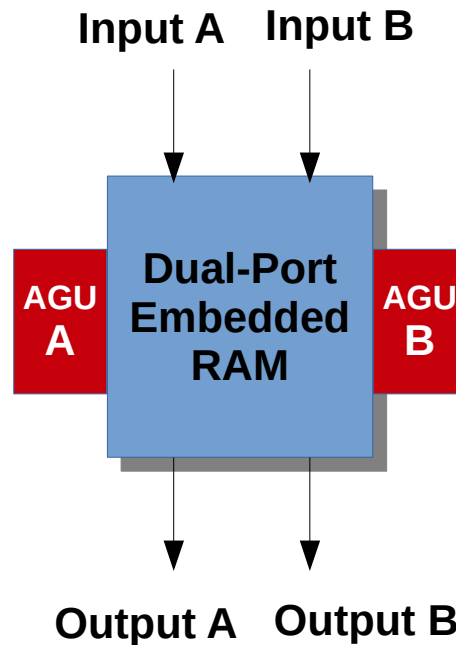
Figure 2.8: Versat Memory Unit with one AGU per port, taken from [18]

2.2.1.2 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.

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 it's 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 it's a write, it stores the data instead.
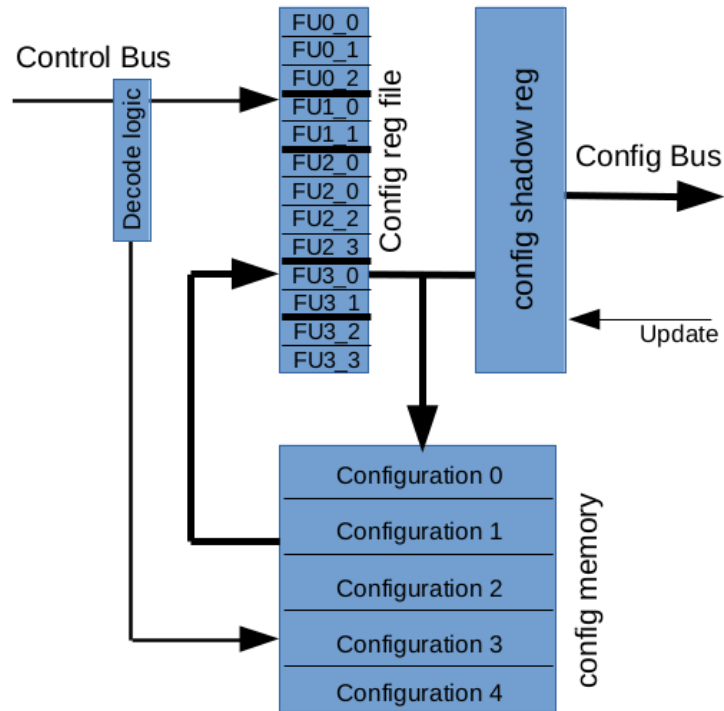
Figure 2.9: Configuration Module,taken from [15]

2.2.2 DeepVersat Architecture

The DeepVersat 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 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 DeepVersat, an API is generated from the Verilog .vh files.
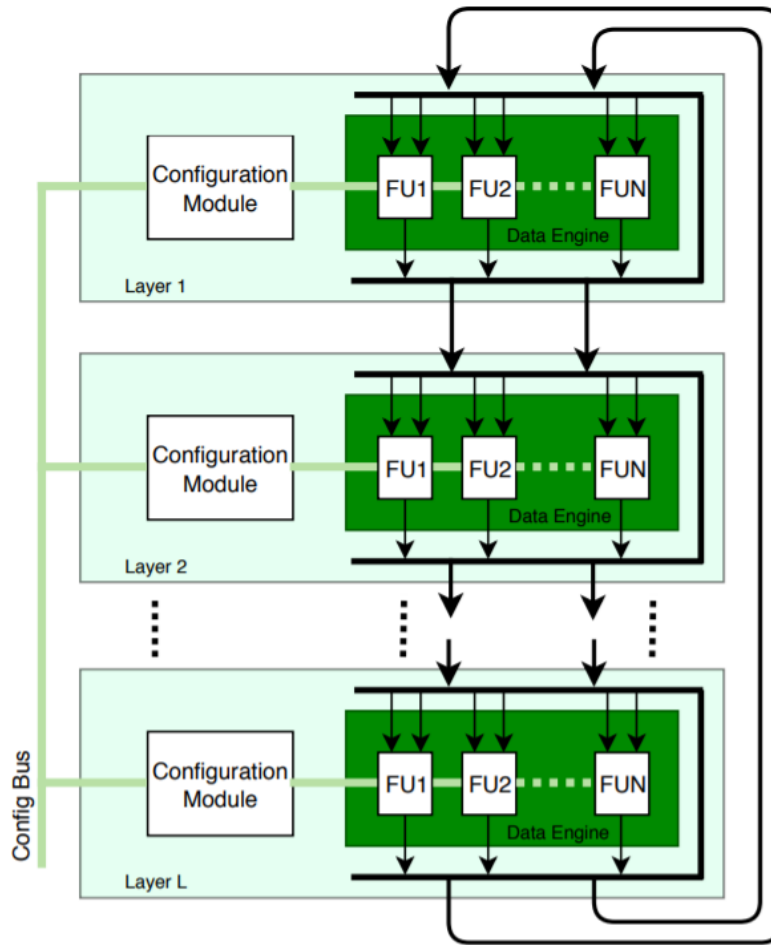
Figure 2.10: DeepVersat Architecture, taken from [1]

2.2.2.1 DeepVersat System
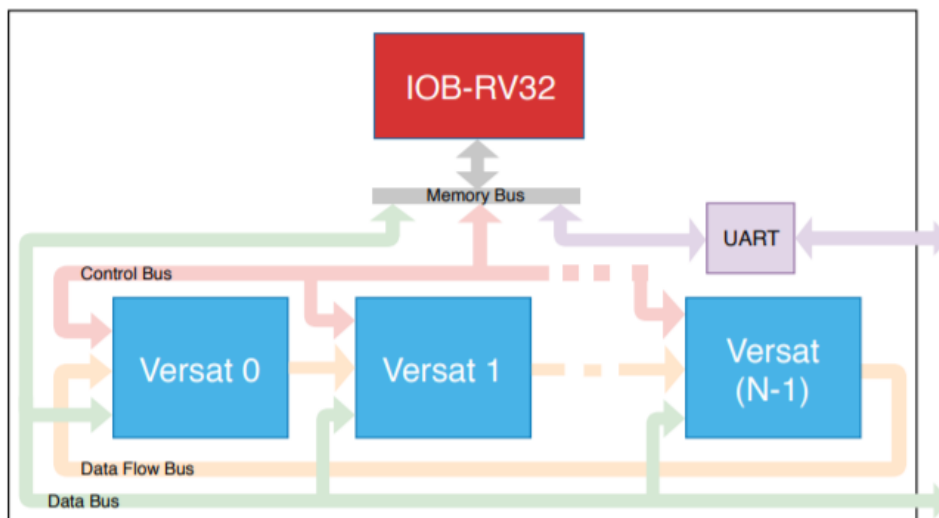
Figure 2.11: DeepVersat System using a RISC-V RV32IMC soft processor, taken from [1]

To make a complete system, a new controller with a more robust toolchain is needed. 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 [19]. The IOB-RV32 uses its memory bus to access peripherals in which DeepVersat and the UART module are connected as such. The control bus is used to access the configuration modules of DeepVersat. The data bus is used to read and write a large amount of data into DeepVersat. The data flow bus is reserved for inter-Versat Core communication.

Peripheral	Memory address
UART module	12'h100xxxxx
DeepVersat control bus	8'h11xxxxxx
DeepVersat data bus	8'h12xxxxxx

Table 2.2: DeepVersat Memory Map

The memory map to address the peripherals, including DeepVersat, is in table 2.2. Each Versat has 15 bits of address while the CPU addresses the peripherals with 32 bits, with eight occupied to choose the peripherals. That leaves nine bits to address several Versat Cores, bringing the theoretical maximum Versat cores to 512. The IOB-RV32 is compatible with the GNU toolchain to offer better code portability, and alongside the C++ Versat API, the difficulty in coding for the System diminishes.

2.3 CNN Compiling in FPGAs

This chapter presents an overview of tool flows 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 provides 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, including throughput and latency in the diversity of applications. Thus, several toolflows that map CNN descriptions into hardware to perform inference have been created. In table 2.3 presents a list of notable ones.

2.3.1.1 Supported Neural Network Models

These toolflows support the most common layers in CNNs, which are discussed in section 2.1. The acceleration target changes depending on the toolflow. For example, the fpgaConvNet [22] tool flow focuses more on feature extraction while offering nonaccelerated support for fully connected layers.

16

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]

2.3.1.2 Architecture & Portability
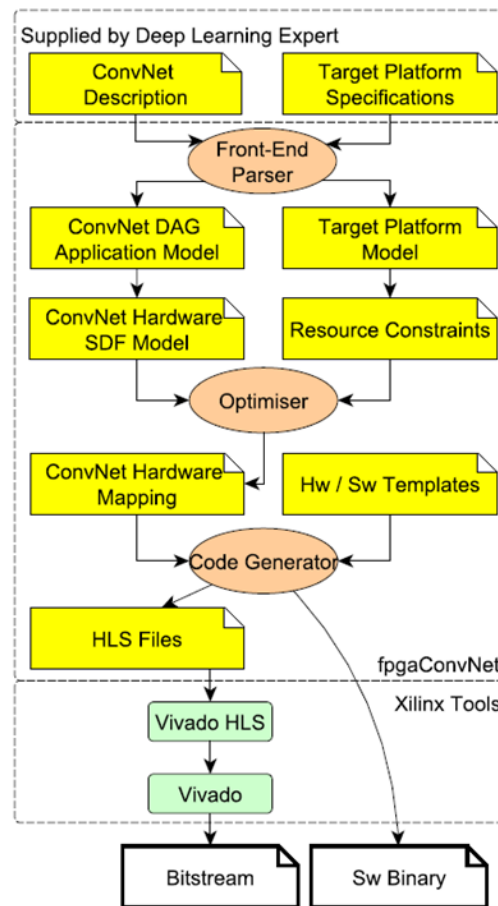
Figure 2.12: fpgaConvNet Architecture. Taken from [22]

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.

Chapter 3

Darknet Lite

The DeepVersat system, as described in Section 2.2, incorporates a RISC-V CPU responsible for executing generic code and storing configuration runs in Versat's memories. This establishes the critical requirement of ensuring compatibility between the embedded CPU and the framework to enable the execution of diverse convolutional neural networks on the system. Furthermore, for optimized performance, the system delegates the execution of fixed functions, such as the convolutional layers, to Versat.

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 prohibited in the embedded code as the RISC-V CPU only supports the extensions 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 appropriately named Darknet lite.

In the following figure, the data structure for a layer is shown. A CNN on Darknet lite is just an array of layers in which each has input, output, and layer parameters. Usually, the input is a past layer output or an 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
```

```c
        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 its definition in C to be run 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 its functions written for CPU or GPU usage.

The following listing 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

```c
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   Parsing CFG Files into the program

Caffe [13] is a deep learning framework as shown in chapter 2, using an open source tool [23], the
output can be set to CFG. By using the network parser of Darknet, an array of layers is created with all
its required parameters.

Listing 3.3: For Loop for writing darknet layers

```
    for( int i=0;i<net->n;i++)
    {
        layer cur=net->layers[i];
        if (cur.workspace_size > workspace_size) workspace_size = cur.workspace_size;
        if (cur.outputs > outputs) outputs = cur.outputs;
        switch (cur.type)
        {
        case CONVOLUTIONAL:write_convolutional_IO(yoloc,i,cur,&base);
            break;
        case CONNECTED:write_connected_IO(yoloc,i,cur,&base);
            break;
        case MAXPOOL:write_maxpool_IO(yoloc,i,cur,&base);
            break;
        case DROPOUT:write_dropout_layer(yoloc,i,cur);
            break;
        case SOFTMAX:write_softmax_IO(yoloc,i,cur,&base);
            break;
        case AVGPOOL:write_avgpool_layer(yoloc,i,cur);
            break;
        case SHORTCUT:write_shortcut_IO(yoloc,i,cur,&base);
            break;
        case ROUTE:write_route_IO(yoloc,i,cur,&base);
            break;
        case RNN:write_rnn_layer(yoloc,i,cur);
            break;
        case YOLO:write_yolo_IO(yoloc,i,cur,&base);
            break;
        case UPSAMPLE:write_upsample_IO(yoloc,i,cur,&base);
            break;
        // Other layers needed to be adressed
        case GRU:
        case CROP:
        case REGION:
        case DETECTION:
```

```
        case COST:
        default :        printf ("\nThis layer is not yet supported.Bye!\n");
                         exit (0) ;
             break;
        }
    }
```

Afterward, by going through each layer, "yolo.c" will be written with all the data darknet lite will need. In listing 3.4, the addresses of the data needed for the layer. In 3.5, the static parameters are defined as well.

Listing 3.4: For Loop for writing darknet layers

```
/*Layer 2−CONVOLUTIONAL*/
#define FOUTPUT_2 BASE+3461616
#define FSCALES_2 BASE+4846064
#define FR_MEAN_2 BASE+4846096
#define FR_VARIANCE_2 BASE+4846128
#define FWEIGHTS_2 BASE+4846160
#define FBIASES_2 BASE+4850768
```

Listing 3.5: For Loop for writing darknet layers

```
/*GENERIC PARAMS−Layer 2*/
[2]. type=0,     [2]. activation =7,    [2]. batch_normalize=1,    [2]. batch=1,
[2]. inputs=692224,    [2]. outputs=1384448,    [2]. n=32,
[2]. h=208,     [2]. w=208,    [2]. c=16,
[2]. out_h=208,     [2]. out_w=208,    [2]. out_c=32,
[2]. size=3,    [2]. stride =1,    [2]. pad=1,
[2]. index=0,     [2]. classes=0,     [2]. total =0,
```

Chapter 4

DeepVersat Software Simulator

The increasing complexity of configurations in the Versat platform and the time-consuming and challenging nature of hardware simulation and debugging have highlighted the need for an efficient software simulator. The primary objective is to emulate the hardware behavior with enhanced efficiency compared to traditional hardware simulation methods. This is particularly crucial as hardware development cycles are significantly longer compared to software development. The simulator operates by executing clock iterations, ensuring consistent results at each clock cycle, similar to the hardware execution. Leveraging the advantages of the Coarse-Grained Reconfigurable Array (CGRA) architecture of Versat, the simulator allows for easy implementation of different functional unit configurations and significantly reduces the time required to assess performance for specific programs. This chapter delves into the software architecture, object relationships, and provides a detailed explanation of the methods employed to emulate Versat clock by clock.

4.1   Architecture and Object Relation

The simulator comprises the Parent Class, Versat, which will be simulated. As each Versat instance is independent of the others, the simulations are also separate. The Versat is made up of two CStage Arrays, one is the "live" while the other is the shadow registers, where the configurations are held before the simulator is run. Each stage is comprised of instances of the FUs defined in the hardware configuration file, each of which is connected to the Databus. As it happens in the hardware, functional units can access the database, which has the output of the current stage and the previous stages' output.

23

Versat

base: int
versat_iter: int
versat_run:int
versat_debug:int
run_done:int

run()
run_sim(void*ie)
done()
versat_init(int base_addr)
globalClearConf()
write_buffer_transfer()
FU_buffer_transfer()
free_mem()

Stages   Shadow Registers

CStage

versat_base:int

start_all_FUs()
update_all_FUs()
output_all_FUs()
copy(Cstage that)
info()
info_iter()
done()
reset()
free_mem()

Databus

CRead

Addr Params

Set addr Params()
start_run()
update()
free_mem()
output()
getdatafromDDR()
AGU()
addrgen2loop_ext()
addrgen2loop()
addrgen4loop()
addrgen6loop()
read(int addr)
copy(CRead that)
copy_ext(CRead that)
info()
info_iter()

CWrite

Addr Params

Set addr Params()
start_run()
update()
free_mem()
output()
getdatatoDDR()
AGU()
addrgen2loop()
addrgen4loop()
write(int addr)
copy(CRead that)
copy_ext(CRead that)
info()
info_iter()

CFU

FU Params

start_run()
update()
output()
copy()
set FU Params()
info()
info_iter()
copy()

Figure 4.1: Class Structure for the Versat Simulator

24

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 creating a new class that CStage will use with a run(), update(), output(), and copy() method. Of course, if it has variables needed to be defined by the program, set param functions are also required. 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. In figure 4.2, a sequence diagram is presented with the rundown of a typical program that uses Versat Simulator.

Program Rundown Diagram

Program   Versat   Simulator

Versat Configurations

Send Configuration

Run()

Shift FU buffer

Shift VO buffer

Run Sim()

Versat Configurations

loop   [Running Loop]

loop   [Each FU]

output()

update()

If VI and VO are over, end Sim

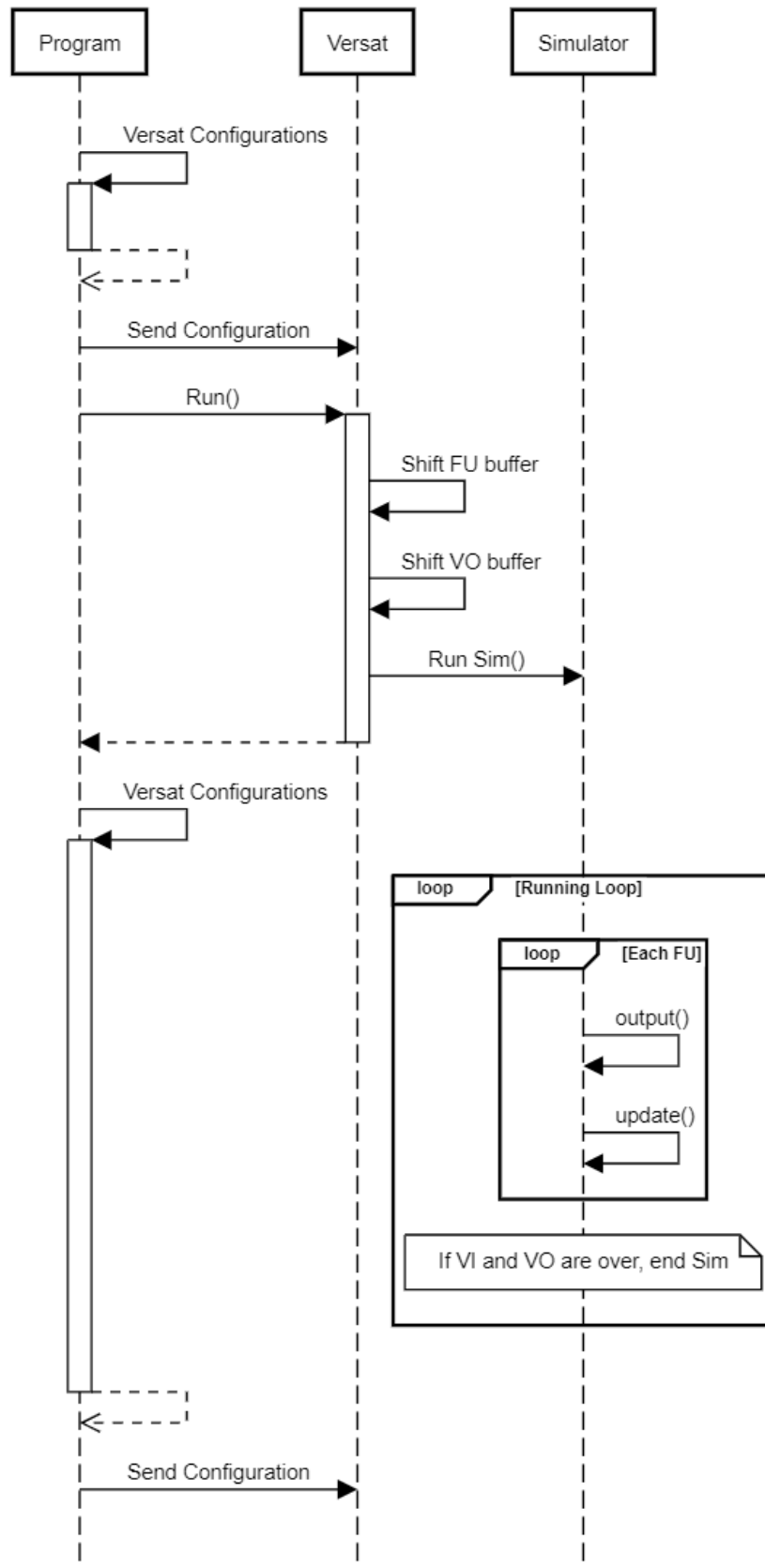Send Configuration

Figure 4.2: Sequence Diagram of a Program using Versat Simulator

26

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 "start", changing the value from zero to 1. Similarly, another register would change the 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 compatibility as the goal is to have the same programs run on software simulators and the FPGA.

Listing 4.1: The Run function code

```cpp
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 the 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 the main memory (DDR), One run cycle is used just for fetching data and writing data. Using a small example: If a developer writes a configuration to do a 5x5 matrix multiplication, Versat will have to run three times. Once to fetch data from memory, the second for the actual use of Versat and the final one is 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 three stages can happen at once if you run multiple configurations in one program, e.g., running a CNN through Versat will have at least one run per layer. So, if it has five layers, Versat will have to run 5+2 times. The last two 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 behavior as the hardware.

4.2.2 Start() Method

At the beginning of the configuration run, the method "start run" of all FUs and memories are started. In this function, several functional units will have their 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 array's data type (versat_t) depends on the width of Versat, which is part of the configuration file. Using higher width, e.g., 64 bits, is useful for the single instruction, multiple data (SIMD) applications but requires the functional units to be adapted. For the purpose 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 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 computing functional units 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

```cpp
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 two functions of the simulator are copy() and info(). The former primary purpose is to copy the configuration parameters from one instance to another, used mainly at 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 complete data of the current iteration, this way, there will be an output file iteration by iteration to check the progress of the simulation, just like in a hardware simulator.

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.

This chapter will discuss the new functions that are part of the Versat API. The goal is to make development for Versat just like writing regular code and to be easy to port code the same way 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

```cpp
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)
```

5.1 API Architecture

Figure 5.1 presents a graphic representation of the new API. It has five apparent layers:

1. Complex Mathematical API that is automatically optimized for the Versat Setup you chose. No dev work is required.

2. Read/Write using VI and VO for simpler data setup. It also includes easier FU functions to set up 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.

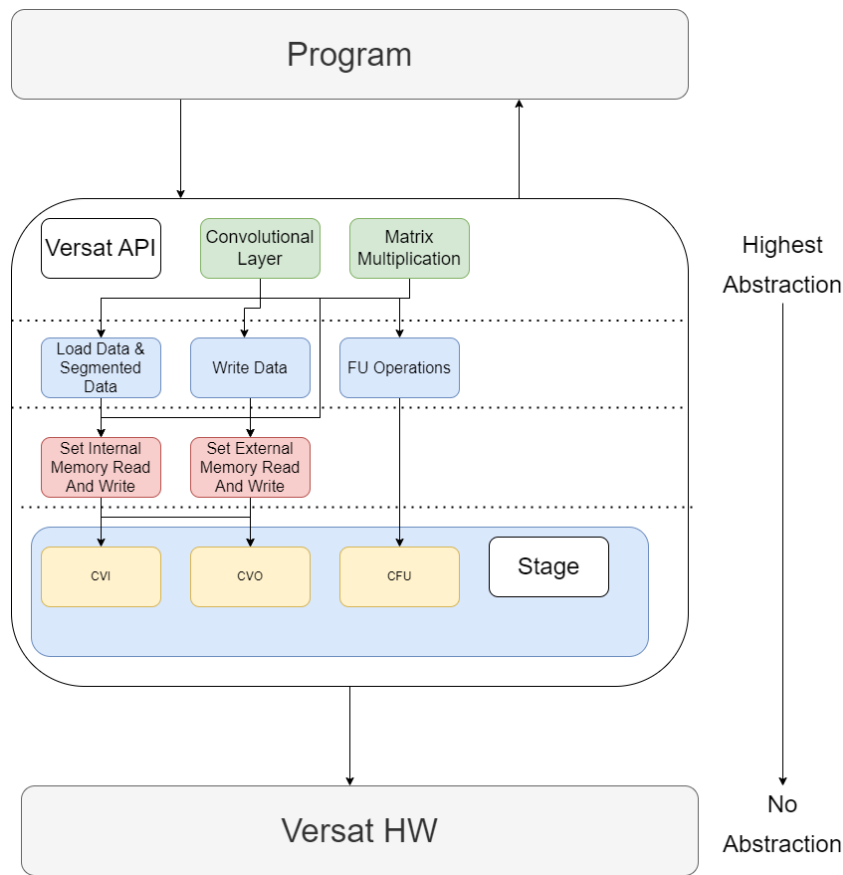Figure 5.1: Graphic representation of the new Versat API and its connections

5.2 Memory Operations API

When utilizing the VI instead of a MEM, the data transfer happens between the functional unit and direct memory access. At the same time, on the mem, the CPU writes directly to Versat, wasting CPU cycles. For the API, this means going from a straightforward read method to more configuration methods to set

up the read operation from DDR. The same happens to Write operations. To address this, seven func-
tions were created in two levels of abstraction: load_data(),load_segmented_data(),write_data() that use a
lower level functions: set_IntMem_Write(),set_ExtMem_Write(),set_IntMem_Read() and set_ExtMem_Read().
The function of the higher abstraction memory functions is to abstract the parameters of the AGU. In the
following listing, we have one of the implementations as an example.

Listing 5.2: Load Segmented Data code

```
int load_segmented_data(CStage* Versat,int index,int addr,int size, int  iter , int  incr)  // size  for  each MEM
{
  Acumulator load;
   int  new_addr=addr;
  load = Acumulator();
  load.add_loop(iter , incr−size);
    load.add_loop(size,1);
    load.loop_settings (0,0,0,new_addr,0);
    set_ExtMem_Read(Versat,index,load);
  return  index+nOUTPUTS;
}
```

Although this means having to write code with the AGUs in mind and how they function. To avoid it,
a new class was created, shown in listing 5.3, to abstract how the AGU counts loops and approximate
the code to simple C++ code that runs on a CPU.

Listing 5.3: Accumulator Class code

```
Acumulator()
{
   iter =per=shift=incr=iter2=per2=shift2=incr2=iter3=per3=shift3=incr3=nloops=delay=duty=start=extAddr=intAddr=0;
}
void  add_loop(int  per, int  incr  = 1 )
{
  switch(nloops)
  {
    case 5:  this−>iter3=this−>per3;
        this−>shift3=this−>incr3;
    case 4:  this−>per3=iter2;
        this−>incr3=shift2;
        if ( this−>iter3==0)
          this−>iter3=1;
    case 3:  this−>iter2=this−>per2;
        this−>shift2=this−>incr2;
    case 2:  this−>per2=iter;
        this−>incr2=shift;
        if ( this−>iter2==0)
          this−>iter2=1;
    case 1:  this−>iter=this−>per;
        this−>shift=this−>incr;
    case 0:  this−>per=per;
        this−>incr=incr;
        if ( this−>iter==0)
          this−>iter=1;
    default :  nloops++;
  }
```

```
}

void loop_settings ( int  start  = 0,  int  duty = 0,  int  delay = 0, int  extAddr = 0,  int  intAddr = 0)
{
  this−>duty=duty;
  this−>start=start;
  this−>delay=delay;
  this−>extAddr=extAddr;
  this−>intAddr=intAddr;
}
};
```

To transform from AGU parameters to for loop, it depends on the number of loops pretended to be done. VI AGU is three cascade Accumulators and as such, the increment on the second and third accumulators needs to be adjusted, as shown below.

Listing 5.4: AGU parameters to Simple forloop parameters transform

```
switch(loop.nloops)
{
  case 6:
  case 5:loop.incr2+=loop.shift*loop. iter +(loop. incr *loop.per)*loop. iter ;  //  4 + 2*2 = 8
        loop.incr3+=loop.shift2*loop. iter2 +(loop.incr2*loop.per2)*loop. iter2 ;
      break;
  case 4:
  case 3:loop.incr2+=loop.shift*loop. iter +(loop. incr *loop.per)*loop. iter ;
   default  :  break;
}
```

5.3 Matrix Multiplication and Dot Product

As part of the new API, a matrix multiplication function was added. The code is presented in listing 5.5. First, two Accumulator class variables are initialized. Afterward, using the two arrays address in DDR, the AGU configurations of the VIs to read from the main memory are set, then the AGU configurations of VI for the data handling inside the Data Engine. Finally, the function will write the MAC configuration and the store AGU configurations. This last step is optional as the result of this matrix multiplication can be used in the same run to make other operations, e.g., adding a bias using one of the ALUs to the results.

Listing 5.5: Matrix Multiplication Configurations

```
int  matrix_mult(CStage* Versat,int matrix_a, int  matrix_b, int  result_matrix , int  r_a , int  c_a, int  r_b , int  c_b,bool store)
{
  Acumulator store_matrix_A= Acumulator();
  Acumulator store_matrix_B= Acumulator();

  //  Send Data from DDR to Versat Memory
  store_matrix_A.add_loop(r_a*c_a);
  store_matrix_A. loop_settings (0,0,0, matrix_a,0) ;
  store_matrix_B.add_loop(r_b*c_b);
  store_matrix_B. loop_settings (0,0,0, matrix_b,0) ;
```

```
set_ExtMem_Read(Versat,0,store_matrix_A);
set_ExtMem_Read(Versat,1,store_matrix_B);

Acumulator read_matrix_A = Acumulator();
Acumulator read_matrix_B = Acumulator();

// Read from Matrix A in Versat
  read_matrix_A.add_loop(r_a,c_a);
    read_matrix_A.add_loop(c_b,−c_a);
      read_matrix_A.add_loop(c_a,1);
        read_matrix_A.loop_settings(0);
        set_IntMem_Read(Versat,0,read_matrix_A);
// Read from Matrix B in Versat
  read_matrix_B.add_loop(r_a,−c_b);
    read_matrix_B.add_loop(c_b,−r_b*c_b+1);
      read_matrix_B.add_loop(r_b,c_b);
        read_matrix_B.loop_settings(0);
        set_IntMem_Read(Versat,1,read_matrix_B);

// Do multiplication of the values and accumulate.
muladd_operation(Versat,sVI[0],sVI[1],0,MULADD_MACC,r_a*c_b,c_a,MEMP_LAT,0);

// Store the results in Memory.
if (store==true)
{
  Acumulator write_matrix = Acumulator();
  write_matrix.add_loop(r_a*c_b,1);
  write_matrix.loop_settings(0,0,MEMP_LAT+MULADD_LAT,result_matrix,0);
  set_ExtMem_Write(Versat,0,write_matrix);
    write_matrix.add_loop(c_a,0);
    set_IntMem_Write(Versat,0,write_matrix,sMULADD[0]);
}



  return sMULADD[0];
}
```

The Dot product function is very similar. The configurations are identical for transferring data from the main memory to the VIs. In the inside loops of the VIs, instead of three loops, we only need to use 1.

5.4   Generic Convolution

As explained in chapter 2, convolutional neural networks are a type of neural net used mostly in image and object recognition using convolutional layers. To run a convolutional layer on Versat with optimized performance, the configurations must be written with regard to several parameters:

1. Memory Sizes used in VI and VO. The amount of data that can be stored at once. It determines the number of outputs done per run.

2. Functional Units used in the Data Engine. Here it's about the lowest common denominator, i.e., the bottleneck in the Data Engine determines the number of outputs done simultaneously.

This function has a total of 20 variables calculated at the start before the Versat configurations are written. The most important variables are the following:

- output height (h) and width (w) of the resulting matrix from the convolution.

- Number of outputs done simultaneously, also known as pipeline width (nOutputs). This value is pre-compiled as it depends on only Versat Configurations.

- Number of outputs that can be done per VI (y) in a single run and its variations. Outputs total (y_2), Output Lines per VI (y_3) Output Lines total (y_4). The value of y4 and y2 decide the different configuration scenarios.

- Resource Allocation Variables which are explained in subsection 5.4.2

- Address Variables

- AGU Configuration Variables

The algorithm's hard part is allocating the data in the most efficient way possible and creating the AGU configurations for the VIs and VOs. For this algorithm, the CGRA will act like a GPU pipeline where several "threads" will exist that will output one point every k^2 cycles, where k is the kernel size used in the convolution.

5.4.1 Loading Data

Usually, when doing a convolution in CPU, the frameworks transform the convolution to a matrix multiplication by creating a new matrix that will multiply with a kernel vector. It's done this way as matrix multiply is a heavily optimized operation and can take advantage of a CPU's SIMD units or even call the GPU APIs and offset the workload there. On Versat, this is not needed to calculate one output. We will need only enough space in mem to hold k^2*ch where ch is the input channel. And as such, it means 9216 bytes per VI, at least for YoloV3 CNN when using 16-bit operands.

To load the data onto the mems in Versat, we will load segmented data. That is, for each mem, we will load the data needed to do y iterations or y_3 iterations, depending on which convolution scenario it is. The more inputs are transferred to a VI mem, the more efficient it is, as data doesn't need to be replicated as much between the instances, i.e., for the first output, there's a need for k^2*ch inputs, but for other sequential outputs, if the stride is one, only k*ch more inputs are needed. But, of course, this is only true if the stride is lower than the kernel size.

This takes the form of the code in one line, thus the importance of the previously written functions.

Listing 5.6: Load Input Matrix into VIs

```
load_segmented_data(stage,i+1,input_addr_new,size_per_channel,channels,in_w*in_h);
```
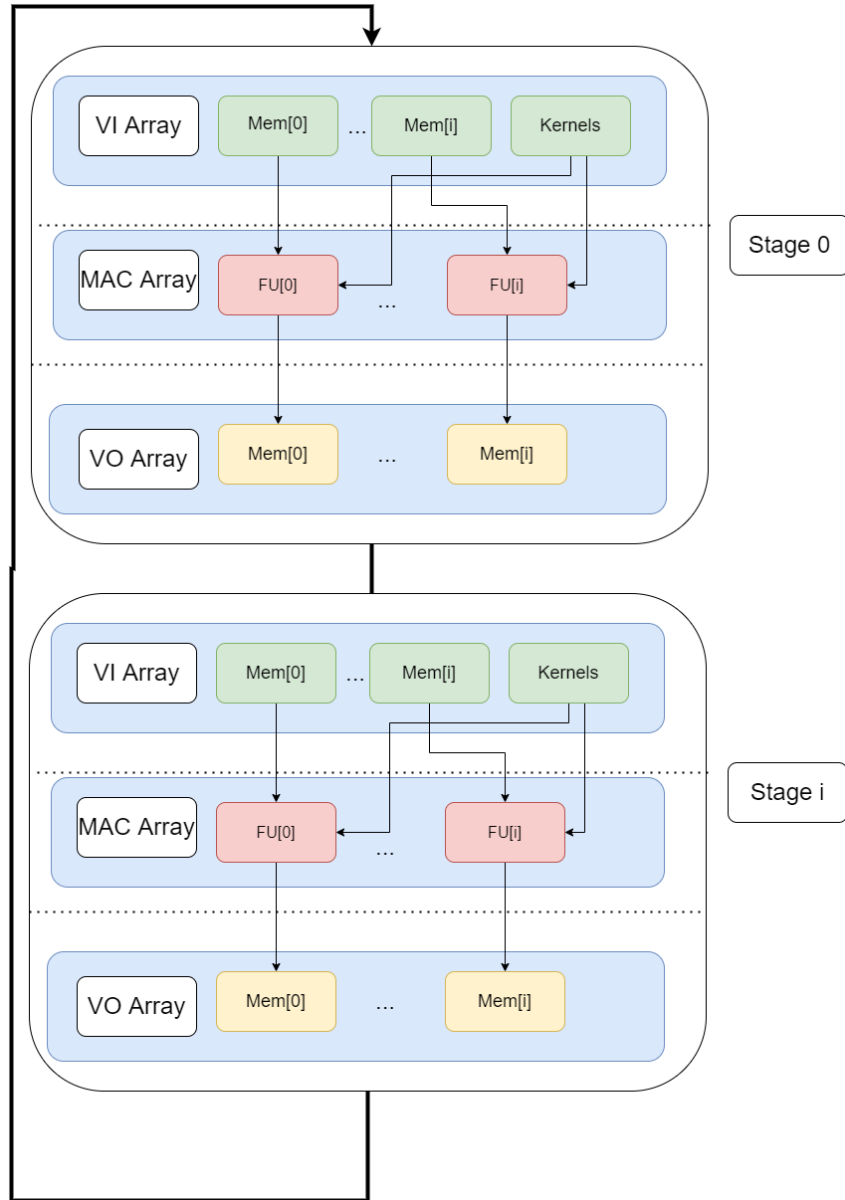
Figure 5.2: Versat Configuration goal in Graphical form

Where the variable "size per channel" can be calculated with the following formula:

$$size = w * (k + stride * (iter - 1))$$

Where w is the width of the input matrix, k is the kernel size, and iter is the number of iterations that this mem will run.

5.4.2 Convolution Scenarios

When writing the configurations of the convolution runs, the software needs to consider several cases. As explained in the previous subsection, the data that the VIs can handle and the number of datapaths that the data can have influenced the convolution scenarios. Four were implemented for this function and presented in figure 5.3.

Where y_2 is the blue squares
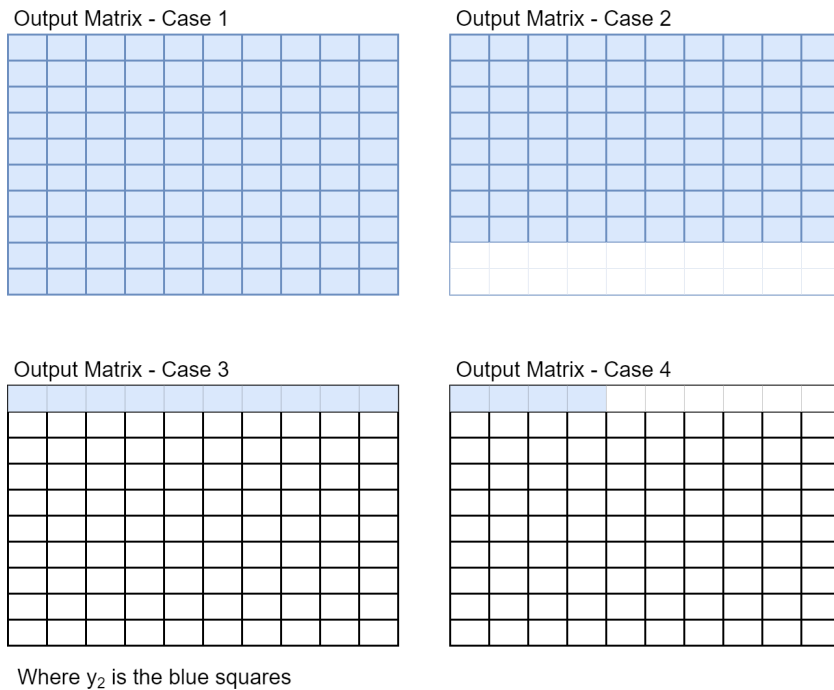
Figure 5.3: Convolution Scenarios that Versat will have

The different hardware configurations and the endless possibilities for convolutions mean that all options are covered. The only limitation of this generic function is to make partial results which is the last case where the mem can't handle enough inputs for one output.

In figure 5.4, the flowchart of each case is presented.
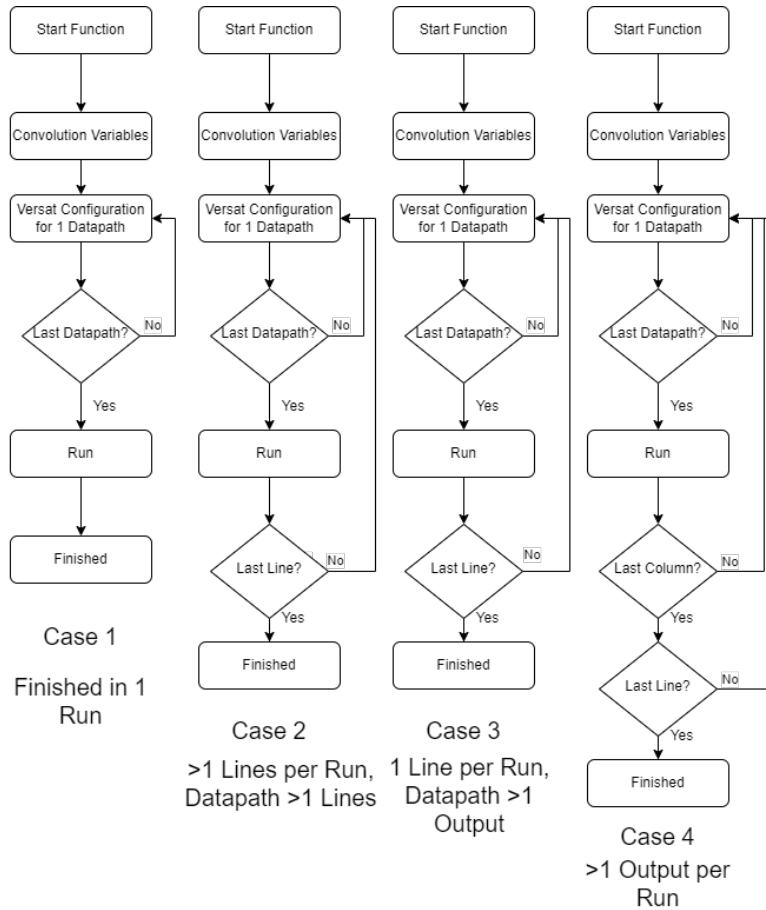
Figure 5.4: Configuration Flowchart for the different scenarios

And on listing 5.7 the AGU configurations of the VIs that hold the input matrix, the MAC configuration, and finally the VO AGU configuration.

Listing 5.7: Versat configurations for one datapath

```
// h1 and h2
input[ i ]= Acumulator();
input[ i ]. add_loop(nkernels,-in_w*((num_iter)*stride));
input[ i ]. add_loop(num_iter,(in_w*stride)-stride*out_w);
  input[ i ]. add_loop(out_w,-channels*size_per_channel+stride);
    input[ i ]. add_loop(channels, size_per_channel-line_plus_one*kernel_size+rewind_kernel);
      input[ i ]. add_loop(kernel_size,line_plus_one);
        input[ i ]. add_loop(kernel_size,1);
          input[ i ]. loop_settings (0);
          set_IntMem_Read(stage,i+1,input[i]);
aux--;

muladd_operation(stage,sVI[i+1],sVI[0], i ,MULADD_MACC,(num_iter)*out_w*nkernels,
kernel_size*kernel_size*channels,MEMP_LAT,0);

write_matrix [ i ] = Acumulator();
write_matrix [ i ]. add_loop(nkernels,out_w*(out_h+1)-(num_iter)*out_w);
write_matrix [ i ]. add_loop((num_iter)*out_w,1);
write_matrix [ i ]. loop_settings (0,0, MEMP_LAT+MULADD_LAT,output_addr_new,0);
set_ExtMem_Write(stage,i,write_matrix[i ]);
  write_matrix [ i ] = Acumulator();
```

```
write_matrix [ i ]. add_loop(nkernels,0);
write_matrix [ i ]. add_loop((num_iter)*out_w,1);
write_matrix [ i ]. loop_settings (0,0,MEMP_LAT+MULADD_LAT,output_addr_new,0);
write_matrix [ i ]. add_loop(kernel_size*kernel_size*channels,0);
set_IntMem_Write(stage,i,write_matrix [ i ], sMULADD[i]);


input_addr_new+=(in_w*(stride*(h1+aux_bool)))*(DATAPATH_W/8);
output_addr_new=output_addr_new+((h1+aux_bool)*out_w)*(DATAPATH_W/8);
```

Chapter 6

Results

In this chapter, experimental tests for Darknet lite, the DeepVersat software simulator, and the new API functions are presented. In Section 6.1, the DNN description of the Darknet Reference Model is translated. Afterward, in section 6.2, the simulator is tested and the results are checked between a CPU-only run and the simulator run. Finally, in section 6.3, test cases for matrix multiplication and generic convolution are presented. The convolution test case features several hardware configurations, which test different simulation scenarios more thoroughly while using a randomized input and kernel.

The tests were executed on a 64-bit machine, with an AMD Ryzen 7 5800H Processor and 16GB of RAM running Windows 11, version 22H2, WSL 2.0 with the image of Ubuntu 20.04. The compiler used is G++ version 9.4.0.

6.1   Compiling DNN Description

The Darknet Reference Model is a 15-layer CNN that is designed to have similar performance to AlexNet**??** while using 1/10th the parameters. It achieves top-1 accuracy of 61.1% and a top-5 accuracy of 83%. In a CPU-only scenario, it takes 0.14s per image. When using the CUDA API, it drops to 2.9ms per image.

To compile the CNN, we run the DNN compiler, which uses the darknet parser to write the configurations for Darknet lite, as mentioned in 3.

Listing 6.1: Darknet Reference Model configuration file for the first four layers

```
[net]
# Training
# batch=128
# subdivisions=1
# Testing
batch=1
subdivisions=1
height=256
width=256
min_crop=128
max_crop=448
```

```
channels=3
momentum=0.9
decay=0.0005

burn_in=1000
learning_rate=0.1
policy=poly
power=4
max_batches=800000

angle=7
hue=.1
saturation=.75
exposure=.75
aspect=.75


[convolutional]
batch_normalize=1
 filters =16
size=3
stride=1
pad=1
activation =leaky

[maxpool]
size=2
stride=2

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

[maxpool]
size=2
stride=2
```

When running the DNN compiler, it checks to see if every layer in the description is implemented in Darknet lite, giving the terminal output in 6.1.

6.2   Simulator Testing

To test the simulator, a test program was created that will create a random input matrix of 5x5 with a kernel size of 3. For each Stage defined in the headers file, a channel will be added, and the result of the convolution will propagate through the stages.

To be more specific, in the beginning, the configurations of the VIs are written to transfer the data from the program to Versat. The data uses the rand() function with seed using current time so the result is different every time. Both the input matrix and kernel map are randomized. The former value

```
jpcardoso@JCLaptop:~/iob-cfg2versat$ ./iob-cfg2versat ./cfg/darknet.cfg
string is .cfg
Parsing .cfg to Versat
layer     filters    size                  input                 output
    0 conv      16  3 x 3 / 1   256 x 256 x   3   ->   256 x 256 x  16  0.057 BFLOPs
    1 max           2 x 2 / 2   256 x 256 x  16   ->   128 x 128 x  16
    2 conv      32  3 x 3 / 1   128 x 128 x  16   ->   128 x 128 x  32  0.151 BFLOPs
    3 max           2 x 2 / 2   128 x 128 x  32   ->    64 x  64 x  32
    4 conv      64  3 x 3 / 1    64 x  64 x  32   ->    64 x  64 x  64  0.151 BFLOPs
    5 max           2 x 2 / 2    64 x  64 x  64   ->    32 x  32 x  64
    6 conv     128  3 x 3 / 1    32 x  32 x  64   ->    32 x  32 x 128  0.151 BFLOPs
    7 max           2 x 2 / 2    32 x  32 x 128   ->    16 x  16 x 128
    8 conv     256  3 x 3 / 1    16 x  16 x 128   ->    16 x  16 x 256  0.151 BFLOPs
    9 max           2 x 2 / 2    16 x  16 x 256   ->     8 x   8 x 256
   10 conv     512  3 x 3 / 1     8 x   8 x 256   ->     8 x   8 x 512  0.151 BFLOPs
   11 max           2 x 2 / 2     8 x   8 x 512   ->     4 x   4 x 512
   12 conv    1024  3 x 3 / 1     4 x   4 x 512   ->     4 x   4 x1024  0.151 BFLOPs
   13 avg                         4 x   4 x1024   ->  1024
   14 conv    1000  1 x 1 / 1     1 x   1 x1024   ->     1 x   1 x1000  0.002 BFLOPs
   15 softmax                                          1000
DBG:*** Printing all internal arrays for this Darknet Model***
DBG:*** Finished Printing all Internal Arrays***
jpcardoso@JCLaptop:~/iob-cfg2versat$
```

Figure 6.1: DNN Compiling of the Darknet Reference Model

varies from -25 to 25, while the kernel varies from -5 to 5. Using the data, we calculate the result of the convolution in the CPU. Afterward, the configuration for the Bias mem is done, and then stage by stage, the configuration of the VI, MAC, and ALU is done. Finally, the configuration of the VO is written.

Listing 6.2: Loading Data into Deep Versat using CMem FU

```
for (j = 0; j < nSTAGE; j++)
{

    // write 5x5 feature map in xread0
    versat.stage[j]. vi [0]. setExtAddr(addr * (DATAPATH_W / 8));
    versat.stage[j]. vi [0]. setIntAddr(0) ;
    versat.stage[j]. vi [0]. setExtPer(25);
    versat.stage[j]. vi [0]. setExtIter (1) ;
    versat.stage[j]. vi [0]. setExtIncr (1) ;
    versat.stage[j]. vi [0]. setExtShift (0) ;
    for (i = 0; i < 25; i++)
    {
        pixels[25 * j + i] = rand() % 50 – 25;
        FPGA_mem[addr] = pixels[25 * j + i];
        addr++;
    }
    versat.stage[j]. vi [1]. setExtAddr(addr * (DATAPATH_W / 8));
    versat.stage[j]. vi [1]. setIntAddr(0) ;
    versat.stage[j]. vi [1]. setExtPer(9);
    versat.stage[j]. vi [1]. setExtIter (1) ;
    versat.stage[j]. vi [1]. setExtIncr (1) ;
    versat.stage[j]. vi [1]. setExtShift (0) ;
    // write 3x3 kernel and bias in xread1
    for (i = 0; i < 9; i++)
    {
        weights[9 * j + i] = rand() % 10 – 5;
        FPGA_mem[addr] = weights[9 * j + i];
        addr++;
```

```
        }

        // write bias after weights of VERSAT 0
        if  ( j  == 0)
        {
            bias  = rand()  % 20 – 10;
            FPGA_mem[addr] = bias;
            versat.stage[j ]. vi [2]. setExtAddr(addr ∗ (DATAPATH_W / 8));
            versat.stage[j ]. vi [2]. setIntAddr(0);
            versat.stage[j ]. vi [2]. setExtPer(1);
            versat.stage[j ]. vi [2]. setExtIter (1);
            versat.stage[j ]. vi [2]. setExtIncr(1);
            versat.stage[j ]. vi [2]. setExtShift(0);
            addr++;
        }
    }
}
```

```
jcardoso13@JPCardoso-Laptop:/mnt/c/Users/joaop/linux_work/thesis/deep-versat/software/pc/testbench$ ./firmware_PC.elf

VERSAT TEST

Deep versat initialized in 374433 us

Data stored in versat mems in 2 us

Expected result of 3D convolution
88      -23     -32
-129    -272    -129
73      51      75

3D CONVOLUTION WITH 4-LOOP ADDRGEN

Configurations (except start) made in 2 us
3D CONVOLUTION WITH 4-LOOP ADDRGEN

Configurations (except start) made in 0 us

Expected Versat Clock Cycles for this run 98

3D convolution done in 391 us
Simulation took 98 Versat Clock Cycles

Actual convolution result
88      -23     -32
-129    -272    -129
73      51      75
```

Figure 6.2: Simulator test output in terminal

The estimated iterations needed are the following:

$$Est = Delay + Iter_2 * Per_2 * Iter_1 * Per_1$$

Where these are the AGU configurations of the VO where the results are written. The Delay is accumulated through the several stages by adding two due to the MACs and ALUs.

Listing 6.3: Writting the configurations of DeepVersat using API v1

```
start  = clock();

// configure mem1B to read bias
versat.stage[0]. vi [2]. setIntStart (0);
versat.stage[0]. vi [2]. setIntIter (9);
versat.stage[0]. vi [2]. setIntPer(9);
versat.stage[0]. vi [2]. setDuty(9);
```

```
for (i = 0; i < nSTAGE; i++)
{

    // configure mem0A to read all 3x3 blocks from feature map
    versat.stage[i].vi[0].setIntIter2(3);
    versat.stage[i].vi[0].setIntPer2(3);
    versat.stage[i].vi[0].setIntShift2(5 – 3);
    versat.stage[i].vi[0].setIntIncr2(1);
    versat.stage[i].vi[0].setIntStart(0);
    versat.stage[i].vi[0].setIntIter(3);
    versat.stage[i].vi[0].setIntIncr(1);
    versat.stage[i].vi[0].setDelay(delay);
    versat.stage[i].vi[0].setIntPer(3);
    versat.stage[i].vi[0].setDuty(3);
    versat.stage[i].vi[0].setIntShift(5 – 3);

    // configure mem1A to read kernel
    versat.stage[i].vi[1].setIntIter(9);
    versat.stage[i].vi[1].setIntIncr(1);
    versat.stage[i].vi[1].setDelay(delay);
    versat.stage[i].vi[1].setIntPer(9);
    versat.stage[i].vi[1].setDuty(9);
    versat.stage[i].vi[1].setIntShift(–9);

    // configure muladd0
    versat.stage[i].muladd[0].setSelA(sVI[0]);
    versat.stage[i].muladd[0].setSelB(sVI[1]);
    versat.stage[i].muladd[0].setFNS(MULADD_MACC);
    versat.stage[i].muladd[0].setPer(9);
    versat.stage[i].muladd[0].setDelay(MEMP_LAT + delay);
    versat.stage[i].muladd[0].setIter(9);

    // configure ALULite0 to add bias to muladd result
    versat.stage[i].alulite[0].setOpB(sMULADD[0]);
    versat.stage[i].alulite[0].setFNS(ALULITE_ADD);
    versat.stage[i].alulite[0].setOpA(in_1_alulite);

    // update variables
    if (i == 0)
        in_1_alulite = sALULITE_p[0];
    if (i != nSTAGE – 1)
        delay += 2;
}

// config mem2A to store ALULite output
// start, iter, incr, delay, per, duty, sel, shift, in_wr
versat.stage[nSTAGE – 1].vo[0].setIntStart(0);
versat.stage[nSTAGE – 1].vo[0].setIntIter(9);
versat.stage[nSTAGE – 1].vo[0].setIntIncr(1);
versat.stage[nSTAGE – 1].vo[0].setDelay(MEMP_LAT + 8 + MULADD_LAT + ALULITE_LAT + delay);
versat.stage[nSTAGE – 1].vo[0].setIntPer(9);
versat.stage[nSTAGE – 1].vo[0].setDuty(1);
versat.stage[nSTAGE – 1].vo[0].setSel(sALULITE[0]);
versat.stage[nSTAGE – 1].vo[0].setExtAddr(addr * (DATAPATH_W / 8));
versat.stage[nSTAGE – 1].vo[0].setExtPer(9);
versat.stage[nSTAGE – 1].vo[0].setExtIter(1);
versat.stage[nSTAGE – 1].vo[0].setExtIncr(1);
```

```
versat.stage[nSTAGE − 1].vo[0].setExtShift(0);
versat.stage[nSTAGE − 1].vo[0].setIntAddr(0);
```

6.3 Testing the new API

In this section, the same method for the previous test file is made. While the previous one relies on using API v1 for the configuration, these test benches run the new API.

6.3.1 Test File for Matrix Multiplication

Figure 6.3: Matrix Multiplication Test File Outputs

The Matrix Multiplication is a quite simple program. The only thing needed is an instance Versat, run versat_init(), create the matrixes, and then use the function matrix_multiplication() The data is also computed in the CPU result to verify the output, which can be found in figure 6.3. In listing6.4, the code of the test file is presented.

Listing 6.4: Writting the configurations of DeepVersat using API v2 for Matrix Multiplication

```
int main(void) {
  int i, j;
  versat_t input_A[4]={1,2,3,4};
  versat_t input_B[6]={5,6,7,8,9,10};
  versat_t expected[6]={21,24,27,47,54,61};
    clock_t start, end;
  uint32_t addr_B=0,addr_res=0;
  printf ("\nVERSAT TEST \n\n");
  CVersat versat;
  start = clock();
    versat.versat_init (VERSAT);
    end = clock();
  printf ("Deep versat initialized in %ld us\n", (end − start));
  printf ("\nExpected result of Matrix Multiplication \n");
  for (i = 0; i < 4; i++)
  {
    FPGA_mem[i] = input_A[i];
    addr_B++;
  }
  addr_res=addr_B;
```

```
for (i = 0; i < 6; i++)
{
   FPGA_mem[i+addr_B] = input_B[i];
   addr_res++;
}

   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 3; j++)
      {
         printf ("%d\t", expected[i*3+j]);
      }
      printf ("\n");
   }

matrix_mult(&versat.stage[0],0,addr_B*(DATAPATH_W / 8),addr_res*(DATAPATH_W / 8),2,2,2,3,true);
// print_versat_config ();
versat.versat_debug=0;

start = clock();
   versat.run();
   while (versat.done() == 0)
      ;
   // print_versat_info ();
   versat.globalClearConf();

   versat.run();
   while (versat.done() == 0)
      ;
int aux_versat_iter =versat.versat_iter ;
   versat.run();
   // print_versat_info ();

   while (versat.done() == 0)
      ;
   end = clock();
   // print_versat_info ();
   printf ("\nMatrix Multiplication done in %ld us\n", (end – start));
   printf ("Simulation took %d Versat Clock Cycles\n", aux_versat_iter);

   // display results
   printf ("\nActual Matrix result \n");
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 3; j++)
         printf ("%d\t",(int16_t)FPGA_mem[addr_res+j+i*3]);
      printf ("\n");
   }
   return 0;
}
```

6.3.2 Test File for Generic Convolution

Using the same method on the previous test benches, the following Convolution Layer was used with several Versat Configurations.

CNN Variable	Value
Kernel Size	2
Channels	2
Number of Kernels	2
Input Height	12
Input Width	12
Stride	1
Out Width	11
Out Height	11
Out Channels	2

Table 6.1: CNN Layer on the test file

With this layer, figure 6.4 has the output result of the generic convolution test file. For this specific Versat hardware configuration, the number of iterations needed is 711 using three Datapaths.


```
jcardoso13@JPCardoso-Laptop:/mnt/c/Users/joaop/linux_work/thesis/deep-versat/software/pc/testbench$ ./firmware_PC.elf
Testing Convolution Layer xyz on Deep Versat
Randomized Input 12x12

Randomized Kernel 2x2

VERSAT TEST

Deep versat initialized in 1141 us
Input ADDR=0
Kernel ADDR=288
Expected Result ADDR=296
Actual Result ADDR=538
Start Test -------------------------
Running Convolution on Versat--------------------------
ENTERED VERSAT CONV
INFO:
h1=3
h2=2
CONVOLUTION CASE - FITS EVERY DATA INTO MEM
CHECK FOR PREVIOUS RUN
VERSAT RUNNING
Versat finished the runs--------------------------
Data Written into Memory--------------------------

Matrix Multiplication done in 64121596 us
Simulation took 711 Versat Clock Cycles

Check Result for Errors:
-30     -4      -26     -180    -26     82      -36     40      176     78      66
22      152     -44     4       98      -14     160     72      -170    -140    -48
136     -120    -16     222     32      -60     -94     -8      128     -30     -192
-156    68      130     110     -64     -106    -106    -152    -160    148     156
-74     -100    -12     -66     58      132     -28     84      2       -56     -16
168     -26     4       90      126     -4      78      -34     152     94      160
-110    54      28      10      92      -34     -162    2       -46     16      -64
-68     -114    56      116     6       -66     -84     -32     4       20      50
-88     130     78      156     24      -72     130     -2      -32     36      -2
192     16      -60     -264    -172    74      22      -56     -60     68      52
6       -26     106     88      82      40      -94     -16     70      80      -70


-311    -24     -137    -36     -115    85      24      192     -14     -83     133
-115    120     -10     -84     -137    187     164     -6      17      -236    166
-120    120     40      97      126     -156    43      206     108     -273    -340
8       114     -61     325     94      -209    -165    -294    102     156     62
-381    14      -92     215     189     -24     -98     -12     -319    112     10
-24     -121    -28     143     75      298     -153    159     38      111     194
139     5       -50     63      274     -21     -59     -127    111     34      54
-206    -85     138     20      197     57      -316    -50     -34     86      81
-76     -91     133     286     80      58      -89     -59     -68     82      -9
-66     114     -130    -84     34      -89     107     -16     -202    202     -16
155     61      199     -70     -133    20      -27     60      -143    126     123
```

Figure 6.4: Generic Convolution test file Outputs

In Table 6.2, the different Datapath numbers and how it affects performance. A datapath is a combination of 1 VI, 1 MAC, and 1 VO. So the lower number in the Versat configuration file decides the number of valid datapaths. Of course, VI needs +1 in numbers more than the functional units due to the Kernel memory.

Number of Datapaths	Iterations
1	1943
2	1063
3	711
4	535
6	359
8	359
11	183
16	183
22	183

Table 6.2: CNN Layer on the test file with several Versat hardware configurations

The reason for these results is quite simple. In total, 11 output lines are divided by the datapaths. When the division is not a whole number, the remainder gets distributed by available datapaths. The consequence of this, when changing from six to eight datapaths, the performance doesn't get any better. Datapath zero will have to run twice to (2 lines) while Datapath 8 will run one line. To increase the performance further, the output channels would have to be divided through more datapaths.

Finally, in Listing 6.5, we present the code for running the generic convolution test, showcasing the reduced code complexity compared to that in Section 6.2. The majority of the code pertains to establishing the testing environment and verifying the results, highlighting the simplified development process for the developer.

Listing 6.5: Writting the configurations of DeepVersat using API v2 for Generic Convolution

```
int main(void) {
  int i, j;
  time_t t;
  srand((unsigned) time(&t));
  cout << "Testing Convolution Layer xyz on Deep Versat\n";

  int kernel_size=2;
  int channels = 2;
  int nkernels = 2;
  int height = 12;
  int width = 12;
  // int height = 9;
  // int width = 9;
  int input_size = height*width*channels;
  int stride = 1;
  int pad = 0;
  int out_w=((width + 2*pad - kernel_size) / stride) + 1;
  int out_h=((height + 2*pad - kernel_size) / stride) + 1;
  cout << "Randomized Input " << height << "x" << width << "\n";
  versat_t input[input_size]={0};
  versat_t kernel[kernel_size*kernel_size*nkernels]={0};


  if (width<10)
  {
    for (i=0; i<input_size; i++) {
      if (i%width==0 && i!=0)
        printf ("\n");
      if (i%(width*height)==0)
```

```cpp
        printf ("\n \n");
      input[i]=rand() % 50 – 25;
      printf ("%d\t",input[i]) ;
    }
    for (i=0; i<kernel_size*kernel_size*nkernels; i++) {
      if (i%kernel_size==0 && i!=0)
        printf ("\n");
      if (i%(kernel_size*kernel_size)==0)
        printf ("\n\n");
      kernel[i]=rand() % 10 – 5;
      printf ("%d\t",kernel[i]) ;
    }
  }
  else
  {
    for (i=0; i<input_size; i++) {
      input[i]=rand() % 50 – 25;
    }
    cout << "\nRandomized Kernel " << kernel_size << "x" << kernel_size << "\n";
    for (i=0; i<kernel_size*kernel_size*nkernels; i++) {
      kernel[i]=rand() % 10 – 5;
    }
  }
    clock_t start, end;
  uint32_t addr_B=0,addr_res=0;
  printf ("\nVERSAT TEST \n\n");
  CVersat versat;
  start = clock();
    versat. versat_init (VERSAT);
    end = clock();
  printf ("Deep versat initialized in %ld us\n", (end – start));
  for (i = 0; i < input_size; i++)
  {
    FPGA_mem[i] = input[i];
    addr_B++;
  }
  int addr_exp=addr_B;
  for (i = 0; i < kernel_size*kernel_size*nkernels; i++)
  {
    FPGA_mem[i+addr_B] = kernel[i];
    addr_exp++;
  }
  addr_res=addr_exp+out_w*out_h*nkernels;
  cout << "Input ADDR=" << 0 << "\n";
  cout << "Kernel ADDR=" << addr_B<< "\n";
  cout << "Expected Result ADDR=" << addr_exp << "\n";
  cout << "Actual Result ADDR=" << addr_res << "\n";
  versat_t acc;
  for( int z=0; z<nkernels; z++)
  {
    for (i = 0; i < out_h ; i++)
    {
      for (j = 0; j < out_w; j++)
      {
        acc=0;
        for ( int n = 0; n < channels; n++)
        {
          for( int l=0; l<kernel_size; l++)
```

50

```
                    {
                         for ( int  k=0;k<kernel_size;k++)
                            {
                               acc+=FPGA_mem[(i*width*stride+j*stride)+(k+l*width)+n*width*height]*FPGA_mem[l*kernel_size+k+addr_B+z*kernel_size*kernel_size];
                            }
                      }
                }
              FPGA_mem[addr_exp+i*(out_w)+j+z*out_w*out_h]=acc;
                if (width<10)
                printf ("%d\t",FPGA_mem[addr_exp+i*(out_w)+j+z*out_w*out_h]);
           }
         if (width<10)
         printf ("\n");
      }
      if (width<10)
      printf ("\n\n");
   }


   // print_versat_config () ;
   cout << "Start Test -------------------------------\n";
   start  = clock() ;
   cout << "Running Convolution on Versat-------------------------------\n";
   versat.versat_debug=0;
   convolutional_layer_xyz (&versat,0,channels,height,width,kernel_size , stride ,pad,addr_B*(DATAPATH_W / 8),addr_res*(DATAPATH_W /
         8),nkernels);
      while (versat.done() == 0)
            ;
      // print_versat_info () ;
      versat.globalClearConf();
      versat.run() ;
   cout << "Versat finished the runs-------------------------------\n";
      while (versat.done() == 0)
            ;
   int   aux_versat_iter =versat. versat_iter ;
      versat.run() ;
      // print_versat_info () ;

      while (versat.done() == 0)
            ;
   cout << "Data Written into Memory-------------------------------\n";
      end = clock() ;
      // print_versat_info () ;
      printf ("\nMatrix Multiplication  done in %ld us\n", (end – start )) ;
      printf ("Simulation took %d Versat Clock Cycles\n", aux_versat_iter) ;

      // display  results
      printf ("\nCheck Result for Errors:\n");
   for ( int  z=0;z<nkernels;z++)
   {
      for  (i = 0;  i  < out_h; i++)
      {
         for  (j  = 0;  j  < out_w; j++)
         printf ("%d\t",(int16_t )FPGA_mem[addr_res+j+i*(out_w)+z*out_w*(out_h+1)]);
         printf ("\n");
      }
      printf ("\n\n");
   }
for  (int  k = 0;  k  < nkernels; k++)
```

```
{
  for ( i=0;i<out_h;i++)
  {
    for ( j=0;j<out_w;j++)
    {
      if (FPGA_mem[addr_exp+k*out_w*out_h+i*out_w+j]!=FPGA_mem[addr_res+k*out_w*(out_h+1)+i*out_w+j])
      {
        cout << "ERROR: Wrong Value at (" << i << "," << j <<","<<k<<")\n";
        cout << "Expected=" << FPGA_mem[addr_exp+k*out_w*out_h+i*out_w+j] << "\n";
        cout << "Actual=" << FPGA_mem[addr_res+k*out_w*out_h+i*out_w+j] << "\n";
      }
    }
  }
}
versat.free_mem();
return  0;
}
```

Chapter 7

Conclusions

In this thesis, a compiler and software simulation model for Deep Neural Networks running on the Deep-Versat Architecture are presented. The simulator runs orders of magnitude faster than an RTL simulator, allowing for the fast testing of new software configurations and workloads. It can accurately predict the performance of the workloads running on DeepVersat. These tools are helpful for architectural exploration, helping to determine the number of functional units, stages, or memory sizes needed for optimal performance.

7.1 Achievements

First, a darknet framework for embedded devices and new tools have been developed to parse CFG, which are essential for future work using the Versat CGRA. These tools make running any CNN on embedded hardware possible, even if it comprises just a CPU.

Second, a software simulation model, referred to as the simulator, has been developed and can emulate the hardware output. A new program was written for Versat can be compiled in seconds instead of the several minutes it takes to compile the DeepVersat FPGA bitstream.

Third, a generic convolution method has been developed to run any convolution layer efficiently. Changing the Versat parameters allows a new hardware convolution configuration to be tested, and the performance can be determined with the simulator.

Lastly, a new Versat API has been developed, which can make writing code for Versat is akin to writing regular C++ code that runs on a CPU.

7.2 Future Work

For future work, prominent sections need to be addressed. For example, while developing darknet lite, they were not linked with Versat and the simulator. For that, a max pool generic function must be added and redirect the convolution layer to the generic convolution for Versat.

Other work includes improving the simulator by adding new FUs and generic functions. On that adding partial results to the convolution will also benefit possible Versat configurations.

Versat is a highly versatile CGRA, but for deep neural networks, datapath width is needed, i.e. more memories and MACs to add more MACs means increasing the propagation time and, as such, grouping VIs and MACs into a bigger functional unit to avoid the usage of a multiplexer at the entrance of the MACs. This could be called the SIMD path, while the rest of the configuration could still be highly configurable and have the standard functional units to have the cake and eat it too. Highest performance and high configurability.

On the memory side, the ability to configure the size of each mem would give more flexibility and the configurations to be more data efficient. For example, in this thesis, there were two types of VIs. One that holds the inputs and another that contains the kernels. The kernels don't use much space, and thus the memory will hold a lot of empty values because both VIs have the same size.

Bibliography

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

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

[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] 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`.

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

[15] 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.

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

[17] 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.

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

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

[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`.

[23] Caffe2darknet python tool. URL `https://github.com/vgsatorras/pytorch-caffe-darknet-convert`.