Deep Neural Networks on the Versat Reconfigurable Processor

João Pedro Costa Luís Cardoso
Electrical and Computer Engineering Department
Instituto Superior Técnico
Lisbon, Portugal
joao.pedro.cardoso@tecnico.ulisboa.pt

***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 Deep Neural Network descriptions into executable code optimized for CPU/DeepVersat system. To achieve this, a neural network framework, Darknet, is extended, adapted, and streamlined to compile neural network 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 Deep Neural Networks 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.**

***Index Terms*—Coarse-Grained Reconfigurable Array, Versat, Darknet, Convolutional Neural Networks, Deep Neural Networks, Simulator, Embedded Systems, Heterogeneous Systems**

I. Introduction

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.

The DeepVersat CGRA is the DNN accelerator to improve the performance of the DNNs in embedded 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.

II. 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 [1], 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 **??** being based on the previous layers, that problem can be eliminated.

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 [2] . 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 [3].

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 [4], 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.

A. Convolutional Neural Networks

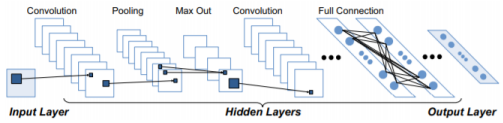

Fig. 1. CNN architecture example, taken from [5]

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 [6] 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.

a) 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 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 1 aided by Figure 2.

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

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.

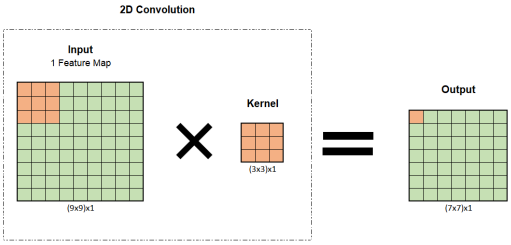

Fig. 2. 2D convolution with stride = one and without zero padding

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 can be used to calculate the output size.

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

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.

b) 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 be also used that turn each pixel into n^2, where n is the number of times the output will be bigger than the input.

c) Fully Connected Layer: The fully connected layer is mostly 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 3 describes the operation.

$$x_j^{l+1} = \delta(\sum_i (x_i^l \times w_{ji}^{l+1}) + b_j^{l+1}) \tag{3}$$

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

d) Route & Shortcut Layer: The Shortcut layer or skip connection was first introduced in Resnet [7]. It allows connecting of the previous layer to another to allow the flow of information across layers. The Route layer, used in Yolov3 [8], concatenates two layers in depth (channel) or skips the layer forward. This is used after the detection layer in Yolov3 to extract other features.

e) Dropout Layer: This type of layer was conceived to avoid overfitting [9] by dropping the neurons with a probability below the threshold.

f) 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 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 I.

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	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$
LReLU	$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$
ELU	$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha e^{x} - 1 & \text{if } x \leq 0 \end{cases}$

TABLE I
POPULAR ACTIVATION FUNCTIONS

B. 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.

1) Darknet: Darknet [10] is an open-source neural network framework written in C and CUDA. It is used as the backbone for Yolov3 [8] 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.

In Listing 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 includes data to be used for training while only the first three parameters are needed for inference.

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

Listing 1. cfg code for a Convolutional Layer used in Yolov3 [8]

2) Caffe: Convolutional Architecture for Fast Feature Embedding (Caffe) [11] 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 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.

III. 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.

A. Versat Architecture

The Versat Architecture [12]–[15] is depicted in Figure 3. 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. Between runs of the Data Engine, the Controller can start doing the next run configuration and calculations.

1) Data Engine: The Data Engine 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

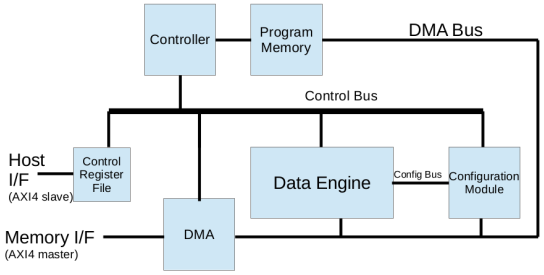

Fig. 3. Versat Topology [13]

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).

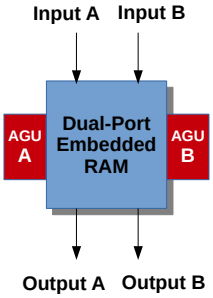

Fig. 4. Versat Memory Unit with one AGU per port [16]

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 results of the operation. Also, the AGUs support delayed start to line up timings due to latencies. The memory module is represented in Fig 4.

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), 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.

B. DeepVersat Architecture

The DeepVersat Architecture [17] , in figure 5, decouples the Data Engine (DE) from all control and as such, it can be

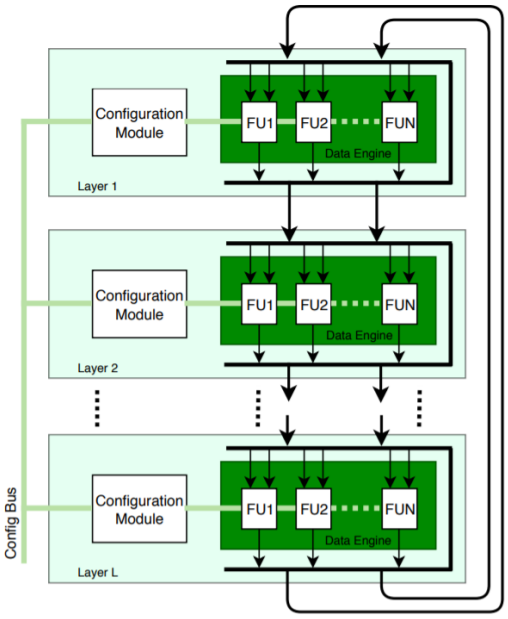

Fig. 5. DeepVersat Architecture [17]

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.

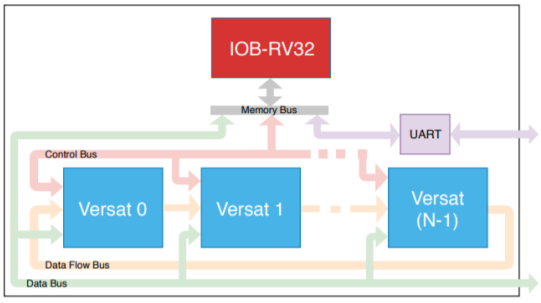

Fig. 6. DeepVersat System using a RISC-V RV32IMC soft processor [17]

1) DeepVersat System: To make a complete system, a new controller is needed with a more robust toolchain. In a recent dissertation [17], 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 [18]. 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.

To address the peripherals, each Versat has 15 bits of address while the CPU addresses the peripherals with 32 bits, with eight of those occupied to choose the peripheral in question. That leaves nine 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.

IV. CNN Compiling in FPGAs

A. Toolflows for Mapping CNNs in FPGAs

Several software frameworks have been developed to accelerate the development and execution of CNNs. The neural networks frameworks discussed in section II-B provides high-level APIs together with high-performance execution on multi-core CPUs, GPUs, Digital Signal Processors (DSPs) and Neural Processing Units (NPUs) [19]. 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 IV-A, a list of notable ones is presented.

Toolflow Name	**Interface**	**Year**
fpgaConvNet	Caffe & Torch	05/2016
DeepBurning	Caffe	06/2016
Angel-Eye	Caffe	07/2016
ALAMO	Caffe	08/2016
Haddoc2	Caffe	09/2016
DNNWeaver	Caffe	10/2016
Caffeine	Caffe	10/2016
AutoCodeGen	Proprietary Input Format	12/2016
Finn	Theano	02/2017
FP-DNN	Tensorflow	05/2017
Snowflake	Torch	05/2017
SysArrayAccel	C	06/2017
FFTCodeGen	Proprietary Input Format	12/2017

TABLE II
CNN to FPGA Toolflows, adapted from [20]

1) Supported Neural Network Models: These toolflows support the most common layers in CNNs, which are discussed in section II-A. The acceleration target changes depending on the toolflow. For example, the fpgaConvNet [21] toolflow focuses more on feature extraction while offering nonaccelerated support for fully connected layers.

2) Architecture & Portability: 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.

V. Darknet Lite

As mentioned in Section III, the DeepVersat 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 implementing software that can run any convolutional neural network on this system, the software must first run on the CPU then we offload Fixed Functions to Versat such as the convolutional layers, max pool, etc.

A. Porting Darknet to an embedded CPU

As mentioned in Section II-B 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.

A CNN on Darknet lite is just an array of layers in which each has input, output, and layer parameters all defined using a layer struct in which all variables are defined. Usually, the input is a past layer output or an image input.

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.

B. Parsing CFG Files into the program

Caffe [11] is a deep learning framework as shown in section IV-A, using an open source tool [22], 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.

VI. DeepVersat 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 being hard to debug.

The goal is to emulate what the hardware is doing much more efficiently than a simple Hardware simulation as the time of development for hardware is much higher than for 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.

A. Architecture and Object Relation

The Simulator is made up of the Parent Class called Versat, which will be simulated itself, as each Versat instance is independent of the others, the simulations are also independent. 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 made up of its Functional Units, of which each 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 one.

1) Functional Units: The following Table contains the functional units present in the simulator. For Memory Units, there are three FUs possible.CMem, VI and VO. CMem reads and writes from the databus between CPU and Versat, while VI and VO are made to communicate through DRAM.

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 or to the left
Memory (Mem)	Sends/Receives data to/from the pipeline. Data is inserted through CPU communication

TABLE III
VERSAT SIMULATOR FUNCTIONAL UNITS

To add a new FU, it’s as easy as creating a new class that will be used by CStage 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 needed. Using the simulator, hardware development and program development can be parallelized to output a new program with more optimized performance.

B. Simulation

After the program that is running on the CPU finishes writing the configurations, it will call the run method of Versat. In figure 7, a sequence diagram is presented with the rundown of a typical program that uses Versat Simulator.

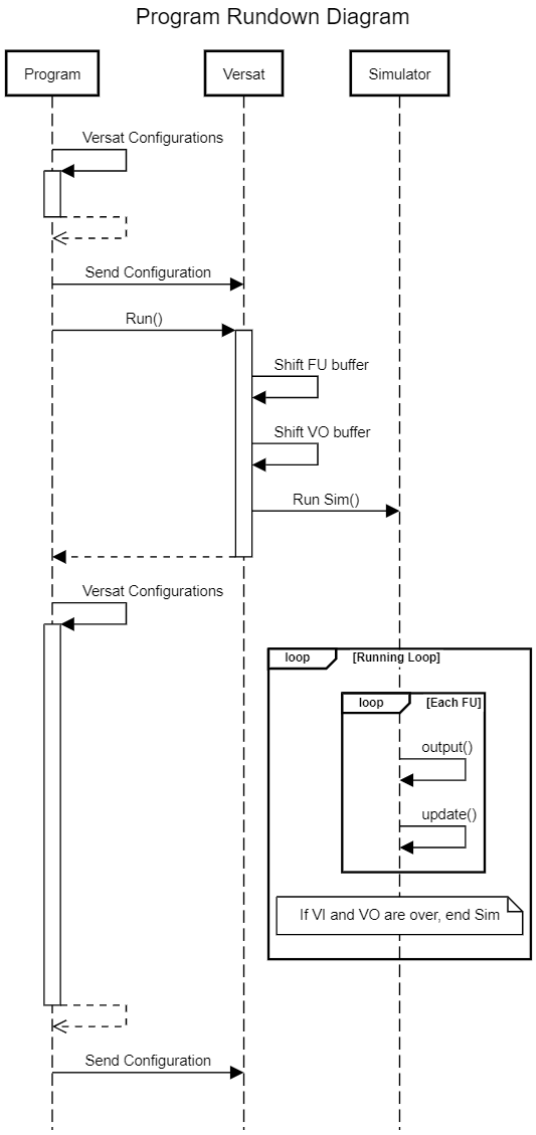

Fig. 7. Sequence Diagram of a Program using Versat Simulator

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.

As can be interpreted from 7, before running the simulation, there is a reset of the state variables, 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), 1 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 simultaneously 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.

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.

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 same 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 it's referenced in the following figure.

As mentioned in figure 5 from section III, 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) 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 memory and performs a read operation. For the write memory, it will output an address and performs a write operation.

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.

VII. Versat API 2.0

The Versat API, developed in a previous thesis [17], can conceal the calls to the hardware to avoid changing the program when the hardware changes.

A. API Architecture

In figure 8, a graphic representation of the new API is presented. It has four apparent layers (5 if you count the hardware):

1) Complex Mathematical 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 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.

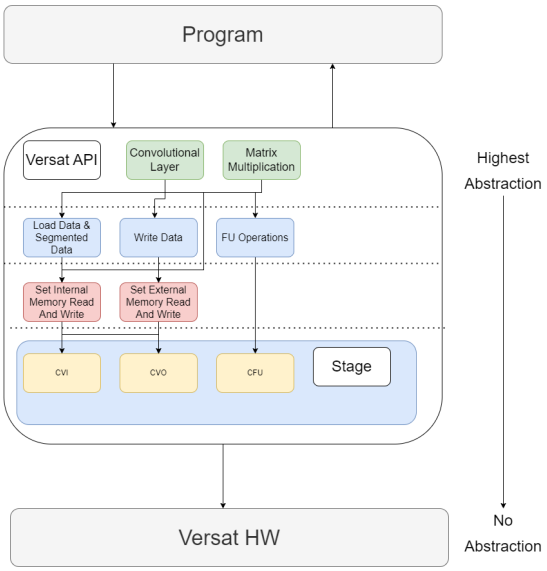

Fig. 8. Graphic representation of the new Versat API and its connections

B. Memory Operations API

When utilizing the VI instead of a standard memory unit, the data transfer happens between the functional unit and direct memory access while in the latter, 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, seven functions 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.

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

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.

C. Matrix Multiplication and Dot Product

As part of the new API, a matrix multiplication function was added. To implement this function, 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 that will write the configuration of a MAC 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.

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

D. Generic Convolution

As explained previously, convolutional neural networks are a type of neural nets that are used mostly in image and object recognition by 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 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 VII-D2
- Address Variables
- AGU Configuration Variables

The hard part of the algorithm is to allocate the data in the most efficient way possible and to create 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.

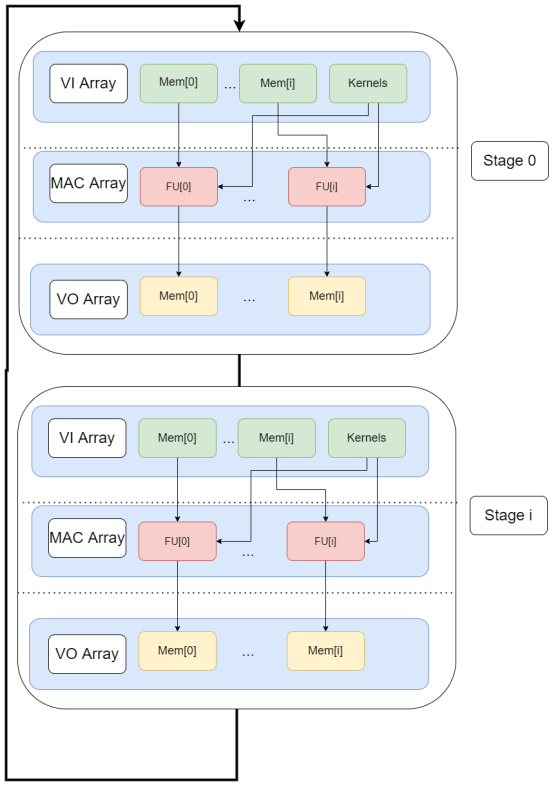

Fig. 9. Versat Configuration goal in Graphical form

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 as to calculate one output, we will need only enough space in mem to hold k^2*ch where ch is the channels of the input. And as such, it means 9216 bytes per VI at least for YoloV3 CNN when using 16-bit operands.

To load the data onto memory in Versat, we will load segmented data. That is, for each memory, 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 unit, 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. of course, this is only true if the stride is lower than the kernel size.

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

```
load_segmented_data( stage , i+1,input_addr_new, size_per_channel , channels ,
    in_w∗in_h);
```

Listing 2. Load Input Matrix into VIs

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.

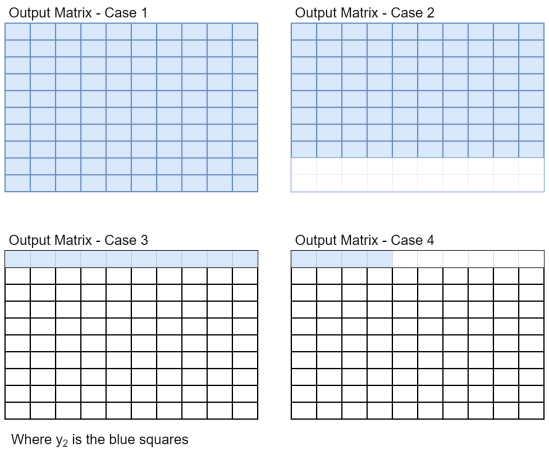

Fig. 10. Convolution Scenarios that Versat will have

2) Convolution Scenarios: When writing the configurations of the convolution runs, there are several cases the software needs to take into consideration. 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. For this function, four were implemented and are presented in figure 10.

The different hardware configurations and the endless possibilities for convolutions mean that all possibilities are covered. The only limitation of this generic function is to make partial results which is the last possible case where the mem can't handle enough inputs for 1 output.

VIII. Results

A. Simulator Testing

To test the simulator, a testbench 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 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.

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.

B. Testing the new API

In this section, the same method for the previous testbench is made. However, while the previous one relies on API v1 for the configuration, these test benches run the new API.

1) Testbench for Matrix Multiplication: 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 as the result to verify the output.

2) Testbench 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 IV
CNN Layer on the testbench

In Table VIII-B2, the different Datapath numbers and how it affects performance. A datapath combines one VI, one MAC, and one 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.

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. Consequently, the performance doesn't get

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

TABLE V
CNN LAYER ON THE TESTBENCH WITH SEVERAL VERSAT HARDWARE CONFIGURATIONS

any better when changing from six to eight datapaths. Datapath zero will have to run twice to (2 lines) while Datapath eight will run one line. The output channels would have to be divided through more datapaths to increase the performance.

IX. CONCLUSIONS

This paper presented several modules and tools for neural networking development on Versat. The simulator significantly improves over standard hardware simulation for testing new software configurations and workloads. It also can predict the performance of the workloads and helps size how many functional units, stages, or how much the size of memories should be to achieve the highest performance. Furthermore, the new Versat API can bring new tools for development using the hardware and be able to write code for Versat akin to writing regular C++ code that runs on a CPU. Finally, the tools designed for Darknet give embedded development a boost by being able to run any CNN on embedded hardware, even if it's just a CPU. To test this, a suite of programs was planned, and the results show the new software tools effectiveness.

A. Achievements

One achievement of this paper was the development of the simulator. The simulator can successfully emulate the output of the hardware, where a new program written for Versat can be tested in five seconds instead of several minutes to put the program into the FPGA. Another achievement is the generic convolution's ability to run any convolution layer efficiently. Changing the Versat parameters allows a new complete hardware configuration to be done and tested with the software to check new performance figures. Lastly, the darknet framework for embedded devices and the tools used to parse CFG are essential for future work using the Versat CGRA.

REFERENCES

[1] Gualtiero 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.

[2] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[3] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993.

[4] mnist database of hand-written digits.

[5] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A cgra-based approach for accelerating convolutional neural networks. pages 73–80, 09 2015.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[8] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

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

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[12] Rui Santiago, João D. Lopes, and José T. de Sousa. Compiler for the versat reconfigurable architecture. REC 2017, 2017.

[13] João D. Lopes, Rui Santiago, and José T. de Sousa. Versat, a runtime partially reconfigurable coarse-grain reconfigurable array using a programmable controller. Jornadas Sarteco, 2016.

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

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

[16] João Dias Lopes. Versat, a compile-friendly reconfigurable processor – architecture. Master's thesis, Instituto Superior Técnico, November 2017.

[17] Valter Jorge Brás Mário. Deepversat: A deep coarse grain reconfigurable array. Master's thesis, Instituto Superior Técnico, November 2019.

[18] Picorv32- a size-optimized risc-v cpu.

[19] Andrey Ignatov, Radu Timofte, Przemyslaw Szczepaniak, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones, 10 2018.

[20] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions, 2018.

[21] Stylianos I. Venieris and Christos-Savvas 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.

[22] Caffe2darknet python tool.