CGRA-Based Deep Neural Network For Object Identification

Pedro José Runa Miranda

Electrical and Computer Engineering

Supervisors:	Prof. Horácio Neto
	Prof. José Teixeira de Sousa

January 2020

Contents

Chapter 1

Introduction

Convolutional Neural Networks (CNNs) are the most commonly used methods to develop accurate object detection and classification algorithms. Some networks divide the problem in two stages: proposing interest regions and perform object localization on each one, like the R-CNN based networks [1–4]. Other approaches like [5–7] use a end-to-end solution that makes predictions directly from the input image. These networks trade some accuracy for speed.

As the CNN models evolve, they become increasingly more demanding in terms of computation and memory accesses. CNNs are trained and deployed on GPUs that achieve a high amount of operations, leveraging time parallelism with Single Instruction Multiple Data (SIMD) architectures, specially effective for processing multiple images (batches). However, these devices also have a significant power demand.

To tackle CNN power demand, there has been a trend to develop hardware dedicated for CNN inference acceleration in reconfigurable hardware architectures such as Field Programmable Gate Arrays (FPGAs), where the power consumption is lower and comes mainly from the memory accesses. These devices allow for the application of a range of optimization techniques.

Coarse-Grained Field Arrays (CGRAs) enable the same parallelism exploration possibilities found in FPGA devices, but at the Functional Unit (FU) level. The higher level datapath manipulation allows for the creation of tool-chains that facilitated the development of accelerators for different networks, lowering the barrier of entry to use such devices.

This work focuses on establishing a background to serve as basis for the development of improvements for the Deep Versat [8] CGRA, enabling the execution of a object detection CNN in real time.

The second chapter provides an overview on neural networks, focussing on inference for the common layers in CNNs. This chapter features an analysis of the YOLOv3 [7] network and Tiny-Yolov3 a smaller version targeting embedded devices. The third chapter outlines the main methods used in CNN acceleration, highlighting datapath optimization and quantization. At the end of the chapter takes place an introduction to CGRAs and analysis of the Deep Versat [8] architecture. The fourth chapter elaborates on the planned work based on the state-of-the-art review performed in the previous chapters.

Chapter 2

CNN Background

Convolutional neural networks (CNNs) were introduced in 1989 by Yann LeCun for digit recognition [9]. Since then CNNs have increased in size and complexity and have gained popularity in the last years for applications like speech processing, robotics and image processing. One example is the performance of AlexNet [10] at the ImageNet Large Scale Visual Recognition Challenge 2012 [11].

In this chapter a background for CNNs is established. The first sections introduce the concepts of inference and training of neural networks. Then, the main types of layers found in CNNs are presented. In the final part of the chapter, an analysis of the YOLOv3 [7] network, a CNN used for image detection, is presented.

2.1 Neural Networks

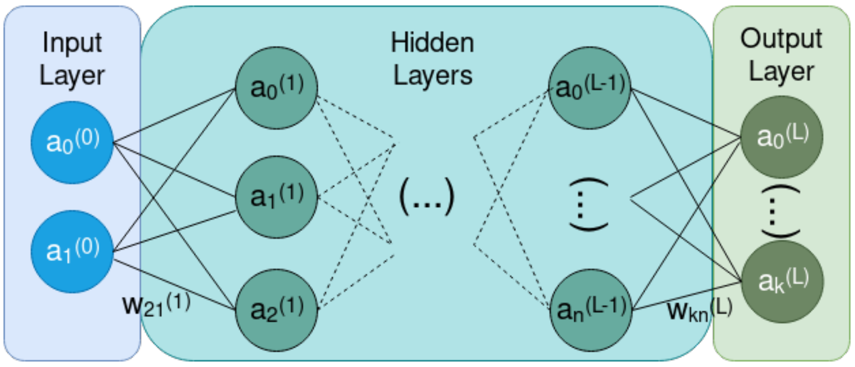


Figure 2.1: Neural Network Structure

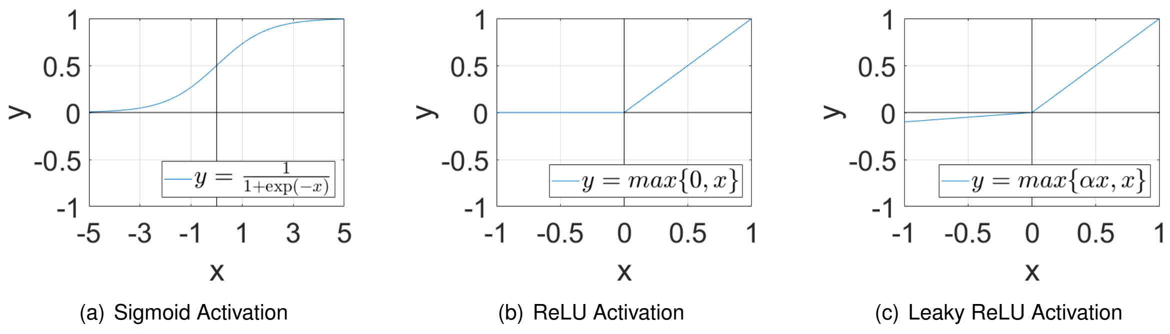
The most basic element of neural networks is the neuron. The output value a of a neuron is given by

$$a = \sigma\Big(b + \sum_{i=0}^{n-1} w_i \times x_i\Big), \tag{2.1}$$

where w_i is the weight associated with input x_i, b is a parameter called bias, $\sigma(.)$ is called the activation function and n is the number of inputs of the neuron.

Fig. 2.1 presents a neural network. Neurons are organized in layers where the neurons in one layer only receive input values from the same set of previous layers. If the input values are the neural network inputs, the layer is called the input layer. The values calculated at the input layer neurons are sent to the next layer. The last layer of the network is called the output layer. The layers in between are called hidden layers.

Activations are non-linear functions applied to each neuron output. The non-linearity of the activation functions allows for multiple layer networks to approximate any function [12]. The sigmoid function, presented in Fig. 2.2(a), is one of the first functions used. In the last years, the Rectified Linear Unit (ReLU), Fig. 2.2(b), and some variations like the leaky ReLU, Fig. 2.2(c), have gained popularity due to the reduced computational complexity during network training.



(a) Sigmoid Activation (b) ReLU Activation (c) Leaky ReLU Activation

Figure 2.2: Plots of common activation functions

2.2 Neural Network Inference

Inference corresponds to the operation of the network over data outside of the training dataset using the already trained weights and biases. The key idea behind this mode is that if the input data being received is similar to the data used for training, then the output generated by the network should reach a comparable accuracy with the one achieved during training.

2.3 Neural Network Training

Neural networks are trained using a training dataset that contains inputs for the network and the desired outputs. During training, the weights and biases of the network are tuned so that given the dataset inputs, the output generated is as close as possible to the desired output. Training is based on gradient descent techniques that update the weights and biases of the network based on the partial derivatives of each value with respect to the difference between the desired output and the one generated by the network. The backpropagation algorithm is the main method used for network training. Training is demanding in terms of computation and memory. The training process is often done in a separate machine, usually taking advantage of the computation capabilities of GPUs.

2.4 Fully-Connected Network

In fully-connected layers, each neuron receives the outputs from all neurons of the previous layer, where there is a different weight associated for each input value. In fact, Fig. 2.1 depicts a neural network composed exclusively by fully-connected layers.

2.5 Convolutional Neural Networks (CNNs)

Unlike fully-connected networks, CNNs are mainly composed by convolutional layers. Other commonly found layers in CNNs are pooling, batch normalization, routing, upsample and shortcut layers.

2.5.1 Convolutional Layer

In convolutional layers, a neuron only depends on a small number of inputs, which corresponds to the neuron only analysing a specific feature for a region of the input. The specific region is called local receptive field. Hence, associated with each neuron, there is only a reduced number of weights and one bias.

In fact, the same local weights and bias are used for a set of neurons, which corresponds to searching for the same feature in the entire input, making convolution invariable to translation in images. The shared weights form a kernel.

The input of convolutional layers is divided into C channels, where each channel is defined as a $(W \times H)$ feature map (FM). For example, in Fig. 2.3 the input has 3 channels and each feature map has size 6×6.

Each kernel used in the convolution has C channels, the same as the input. The number of output channels is the same as the number of kernels used. Each value on the output is the accumulation of the products of the input with the overlapped kernel.

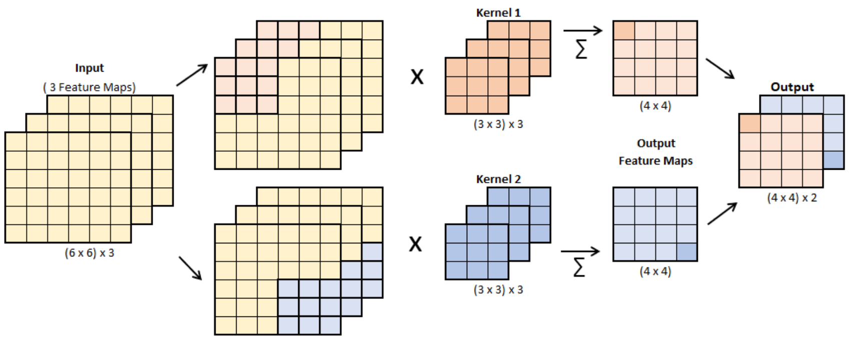


Figure 2.3: Example of a 3D convolution

2.5.2 Pooling Layer

CNNs commonly use pooling layers to reduce the size of the feature maps. Intuitively this corresponds to only keeping a rough idea of the feature location. The reduction of the feature maps also reduces the amount of computation at latter layers of the network. The most common pooling operations divide the feature map into 2×2 regions and select either the larger (Max-pooling) or the average (Average-pooling) value.

2.5.3 Batch Normalization Layer

Batch normalization sets the average of the input values to zero and the standard variation to one. After that, the values are scaled and shifted using the (γ,β) parameters also learned in training. This control of the input distribution speeds up training and improves accuracy [13]. For a given value x, the normalization outputs

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2.2}$$

Since all variables are known after training, is possible to rewrite (2.2) as

$$\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}x + \left(- \frac{\mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \right) = Ax + B, \tag{2.3}$$

which avoids the computation of the square root and division. The additional ϵ term is used to avoid numerical errors related with denominators too close to zero.

2.5.4 Routing Layer

Routing layers are used to get an output from a previous layer in the network. If multiple previous layers are selected, their output is concatenated.

2.5.5 Upsample Layer

Upsample layers increase the size of the input FMs, doubling their width and height. The simplest upsample method consists in repeating each value in a FM four times in a 2×2 square.

2.5.6 Shortcut Layer

Shortcut layers add the output from two or more previous layers in the network pipeline. Shortcut layers are used to mitigate the vanishing gradient problem that arises during training for networks with considerable depth. With this addition, the networks act as already knowing the result from the further back layer and only need to learn a residual to get the desired outcome. Shortcut layers are common on ResNet networks [14].

2.6 YOLO

YOLO (You Only Look Once) [5–7] is an object detection and classification system that is composed of a single CNN that receives an image and outputs bounding box coordinates and class probabilities from the detected objects.

A YOLO network is composed of a sequence of convolutional layers that serve as feature extractors at different scales. During these layers of the network, the size of the FMs is gradually reduced whether by pooling layers or convolutional layers with stride 2, depending on the network version.

After the feature extraction, object detection and classification is done at different FM sizes, which corresponds to analysing the input image at different $g_i \times g_i$ grids.

For each cell in each grid, M attempts of object detection are done. Each attempt is represented by two values for the object box center, two for the object box size, one value for the objectness score and 80 class scores. The objectness score indicates the likelihood of the detection corresponding to an object. The 80 class scores correspond to the number of existing classes in the COCO dataset [15] used for training. In total an object detection attempt is represented by $(2+2+1+80) = 85$ values.

Before the object detection and classification at grid size $g_i \times g_i$ starts, a set of $(M \times 85)$ FMs of size $g_i \times g_i$ is created, and each detection value is associated with a specific FM.

The object detection and classification process uses a custom type of layer called yolo layer. At the yolo layers, the FMs of the channels corresponding to the box coordinates, objectness score and classes scores are passed through a sigmoid activation function (Fig. 2.2(a)). The channels for each output type and the activation performed at the yolo layer is presented in Tab. 2.1, for the case of $M = 3$. The outputs of the several yolo layers correspond to the outputs of the network.

Channel Ranges	Outputs	Activation
$\{[0,1];[85,86];[170,171]\}$	Box center coordinates	Sigmoid
$\{[2,3];[87,88];[172,173]\}$	Box size	None
$\{4;89;174\}$	Objectness score	Sigmoid
$\{[5-84];[90-169];[175-254]\}$	Class scores	Sigmoid

Table 2.1: Yolo layer channel composition and performed activations for $M = 3$.

After all the outputs of the network are obtained, all the detections done by the network need to be filtered. This consists in excluding detections with an objectness score below a determined threshold; and then applying Non-maximum Suppression (NMS) to eliminate multiple detections of the same object. The NMS process starts from the detection with highest objectness score and resorts to a Intersection over Union (IoU), illustrated in Fig. 2.4, between the current detection box and all the others. If the result is above the IoU threshold, the detection with the lowest objectness score is discarded, otherwise it is kept. This is done for all remaining detections in a descending order of objectness score.

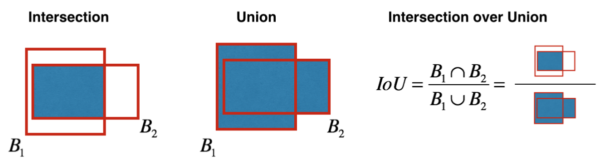


Figure 2.4: Diagram of the Intersection over Union (IoU) calculation, from [16]

2.6.1 Accuracy Metrics

The mean Average Precision (mAP) is a measure used to compare networks that use the Common Objects in Context (COCO) dataset [15].

The mAP is the average of the AP calculated for each class. The AP is calculated by ordering the classifications performed by the highest bounding box confidence level. The classifications are evaluated as true positives or false positives according to the ground truth. At each classification is also calculated the precision and recall, given by

$$recall = \frac{\#True\ Positives}{\#TotalTruePositives} \text{ and } precision = \frac{\#True\ Positives}{\#Classifications}. \tag{2.4}$$

The true positives are the number of correct classifications up to the classification being analysed and the total true positives are the total number of objects in the image. The number of classifications corresponds to the order of the classification being analysed. A precision/recall curve is calculated with the values computed for each classification as is presented by the orange curve in Fig. 2.5. The curve is then altered in order to have monotonically decreasing precision, by setting the precision at a given point equal to the maximum precision for any value of higher recall, as is presented by the green curve in Fig. 2.5. Finally, the AP is calculated by computing the area under the green curve.

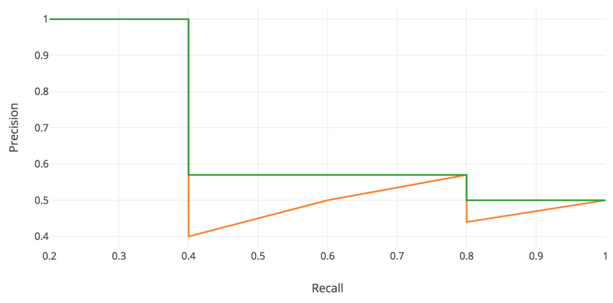


Figure 2.5: Example of precision/recall curve, adapted from [17]

Tab. 2.2 presents the mAP for a 0.5 IoU threshold and respective inference time (in ms) for multiple networks. The results were obtained using machines with an M40 or Titan X Pascal GPUs [7] as indicated. The results show that the Yolov3 networks achieve an accuracy on pair with the best performing networks at a fraction of their inference time. The different versions of the Yolov3 networks are related with the resolution of the input images, but all have the network structure.

Network	mAP_{50}	Time (ms)	FPS	GPU
SSD321	45.4	61	16.4	M40
DSDSD321	46.1	85	11.8	M40
R-FCN	51.9	85	11.8	M40
SSD513	50.4	125	8	M40
DSSD513	53.3	156	6.4	M40
FPN FRCN	**59.1**	172	5.8	M40
RetinaNet-50-500	50.9	73	13.7	M40
RetinaNet-101-500	53.1	90	11.1	M40
RetinaNet-101-800	57.5	198	5.1	M40
Yolov3-320	51.5	**22**	**45.5**	Titan X Pascal
Yolov3-416	55.3	29	34.5	Titan X Pascal
Yolov3-608	57.9	51	19.6	Titan X Pascal

Table 2.2: mAP and inference time comparison between multiple networks, adapted from [7, 18].

2.6.2 Yolov3

As presented in Tab. 2.2, the Yolov3 network performs object detection at a higher framerate than other networks, while maintaining a comparable level of accuracy.

Layer Type	Number of Layers
Convolutional	75
Shortcut	23
Route	4
Yolo	3
Upsample	2
Total	107

Table 2.3: Yolov3 layer type composition.

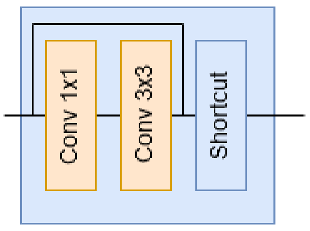
Tab. 2.3 presents the layer composition of the Yolov3 network. The majority of the layers are convolutional. Most of the convolutional layers are used for feature detection or for reducing the FM resolution. This network uses convolutional layers with 3×3 kernels with stride 2 instead of pooling layers.

Due to the depth of the network, shortcut layers are used, to tackle the problems described in section 2.5.6.

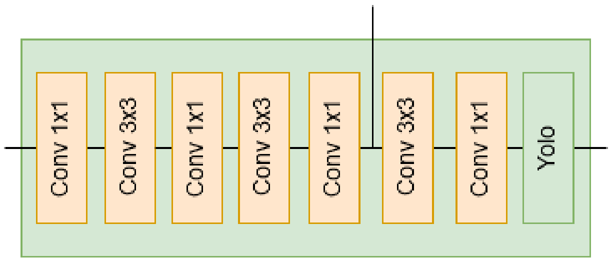
The accuracy performance for small object detection also comes from using three different cell grid scales as opposed to the two used in previous versions [6]: (13×13), (26×26) and (52×52). The indicated resolutions are specific to the Yolov3-416 version.

Fig. 2.6 presents the main stages that constitute recurring sequences of layers in the network: the F stage presented in Fig. 2.6(a) and the Yolo stage presented in Fig. 2.6(b).

Each F stage is composed of a convolutional layer with 1×1 kernels followed by another with 3×3 kernels. At the end of the stage there is a shortcut layer that adds the outputs of the second layer to the input of the stage.



(a) F stage composition



(b) Yolo stage composition

Figure 2.6: Stages of Yolov3 Network

In each Yolo stage, there is a sequence of convolutional layers, with 3×3 and 1×1 kernels that generate FMs of size 13×13 into the first yolo layer. Since Yolov3 performs three detection attempts per grid cell, there are $3 \times 85 = 255$ FMs as input for every yolo layer. The classification takes place for a grid with the same size as the FMs and the number of FMs corresponds to the number os parameters to be calculated per grid cell.

A complete scheme of the network is presented in Fig. 2.7.

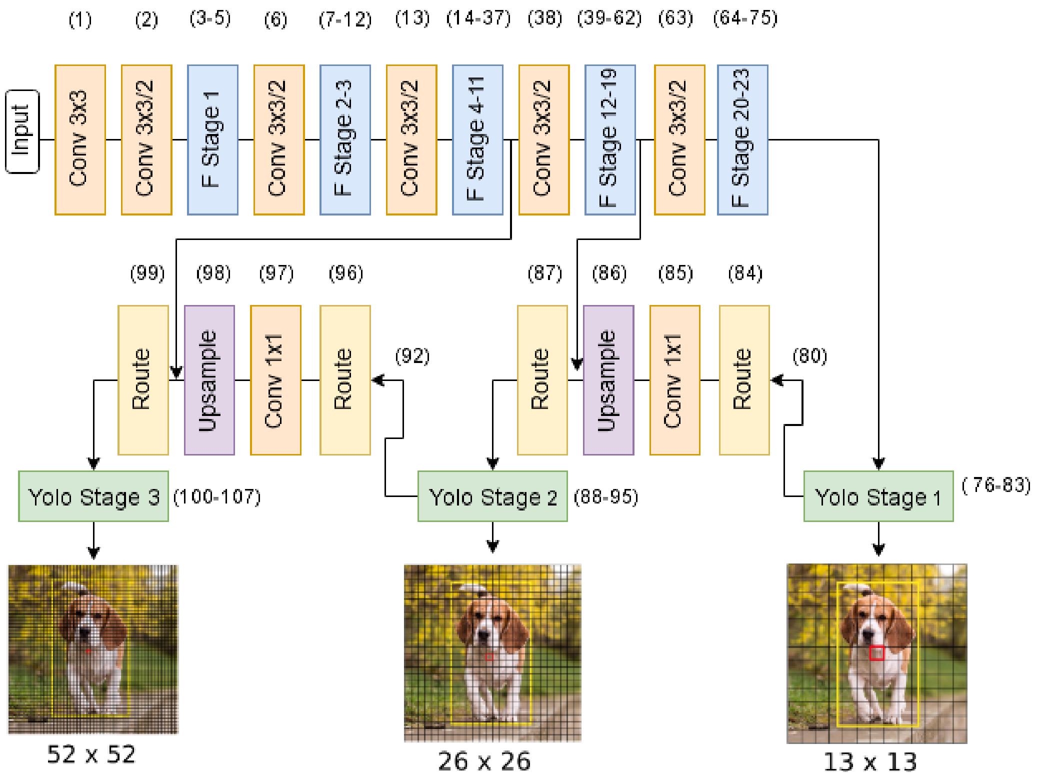


Figure 2.7: Diagram of Yolov3 network

The first part of the network is composed of a series of convolutional layers terminated by F stages. Convolutional layers with stride 2 reduce the FMs by a factor of four along the way. In this network implementation, the convolutions use zero padding around the input FMs, so the size is maintained in the output FMs. This part of the network is responsible for the feature detection in the input image.

The second part of the netwok begins with a Yolo stage after which there is a route layer that concatenates the output of the layers indicated in Fig. 2.7. This output is then compacted in terms of depth by a 1×1 convolution and upsampled by a factor of 2 in each FM dimension, preparing the FMs for detections in a grid of 26×26 cells and consequently detecting smaller objects in the image. These FMs are then concatenated with the set of FMs of the same size from the detection part of the network. The combination of the outputs of these two layers contribute with more meaningfulness using the information from the upsampled layer and the finer-grained information from the earlier feature maps [7].

After condensing the depth of the layer output to 255 FMs (again with a convolutional layer with 1×1 kernels), a second Yolo stage is used for object detection at the new scale. The process is repeated one more time for the third detection at the last yolo stage for a 52×52 grid.

2.6.3 Tiny-YOLOv3

For constrained environments, a small version of YOLOv3 is available. The layer composition of the network is presented in Tab. 2.9.

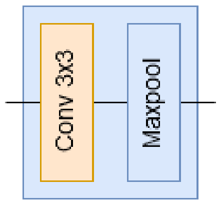


Figure 2.8: Tiny-Yolov3 stage composition

Layer Type	Number of Layers
Convolutional	13
Maxpool	6
Route	2
Yolo	2
Upsample	1
Total	24

Figure 2.9: Tiny-Yolov3 layer type composition.

The first part of the network is composed of repeated Tiny-Yolov3 stages presented in Fig. 2.8. Each stage is composed of a convolutional layer with 3×3 followed by a maxpooling layer.

This network follows the same sequence of layers present in Yolov3. The main differences are in the number of layers and in the number of detections performed. The Tiny-Yolov3 network performs object detection at only 2 grid resolutions: 13×13 and 26×26.

Final Remark

State-of-the-art CNN models are widely used for object detection applications. These networks rely on convolution to extract features from the images in order to be able to perform object detection. In order to achieve higher performance metrics, CNNs grow in complexity. The Yolov3 [7] network obtains comparable mAP for the COCO dataset [15] to other networks, at a fraction of the inference time.

Chapter 3

CNNs in Reconfigurable Hardware

CNNs have been implemented on multiple devices, from CPUs and GPUs to FPGAs. CPUs and GPUs mostly use temporal architectures, while FPGAs, which are in the class of reconfigurable hardware devices use spatial architectures to achieve parallelization. Another type of reconfigurable hardware devices is the Coarse Grained Reconfigurable Array (CGRA) architecture, which, despite powerful, is not as studied as FPGAs for the implementation of CNNs, and is the main focus of this work.

An effective parallelization is specially important in the convolutional layers which are responsible for 90% of the execution time during inference [19].

This chapter focusses on the main techniques used to accelerate convolutional layers in Reconfigurable Hardware. First, the inherent parallelisms of the convolutional algorithm are highlighted, followed by a presentation of the main optimization techniques used. Closing the chapter, the Deep Versat CGRA, which will be used in this work, is reviewed.

3.1 General CNN Algorithm

The data used in a convolution is presented in Fig. 3.1. A particular layer has an input X composed of C FMs of size ($W \times H$) and uses N kernels (Θ_1 to Θ_N) of size ($J \times K \times C$) to obtain the output Y composed of N FMs of size ($V \times U$). The generic CNN algorithm is presented in Alg. 1. The bias is not represented for simplicity.

```
for $n \in \{1, \ldots, N\}$ do // Loop 4 iterates over the $N$ channels of output $Y$
  for $v \in \{1, \ldots, V\}$ do
    for $u \in \{1, \ldots, U\}$ do // Loop 3 iterates over the pixels within the same output FM
      for $c \in \{1, \ldots, C\}$ do // Loop 2 iterates over the channels of input $X$
        for $k \in \{1, \ldots, K\}$ do
          for $j \in \{1, \ldots, J\}$ do // Loop 1 iterates over a 2D kernel window
            MACC: $Y[u, v, n] += X[j, k, c] \times \Theta_n[j, k]$
```

Algorithm 1: General CNN algorithm for a single layer.

The commented loops are candidates for loop unrolling, as discussed in section 3.3.1.

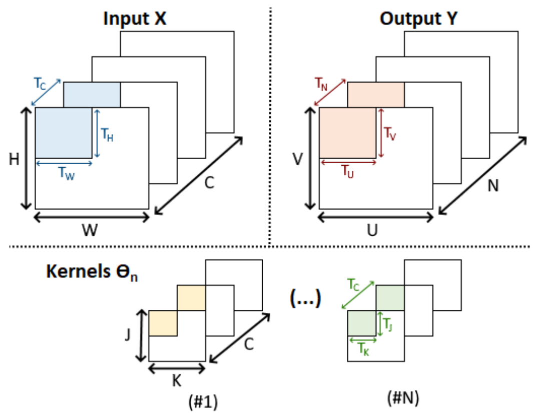


Figure 3.1: Input pixels, kernels and output pixels representation, adapted from [19]

3.2 CNN Inference Acceleration in FPGAs

In general, acceleration for CNN inference in FPGA is done by a combination of datapath and CNN model optimizations.

Datapath optimizations explore parallelism opportunities in the convolutional algorithm described in 3.1. However due to resource limitations in the devices when compared with the amount of calculations done in a single convolutional layer, only some parallelisms can be explored.

The last optimization type consists in trading model accuracy to improve computational and energy efficiency. This can be achieved by operating over reduced precision operands or reducing the model size.

3.3 Datapath Optimizations

In [19], SIMD accelerators are presented as the main strategy for implementing datapath optimizations. The architecture of a generic SIMD accelerator is presented in Fig. 3.2. SIMD accelerators receive the input FMs and weights from external memory into an on-chip buffer. This data is then processed in a pool of PEs that output the results into an output buffer. The output is then sent back to the external memory. Each PE contains on-chip registers and performs the computations using DSP blocks. The two levels of caching implemented by the on-chip buffers and registers reduce the accesses to external memory. This reduction has a significant impact in terms of power consumption, since the accesses to external memory are the determinant factor for power consumption in FPGAs [13, 19].

With this accelerator architecture, the process of datapath optimization consists in finding the configuration of the PEs and data schedule that maximizes computational throughput, specially for convolutions.

Since the convolution is a set of nested loops (Alg. 1), loop optimization techniques as loop unrolling, loop tiling and loop interchange are applied.

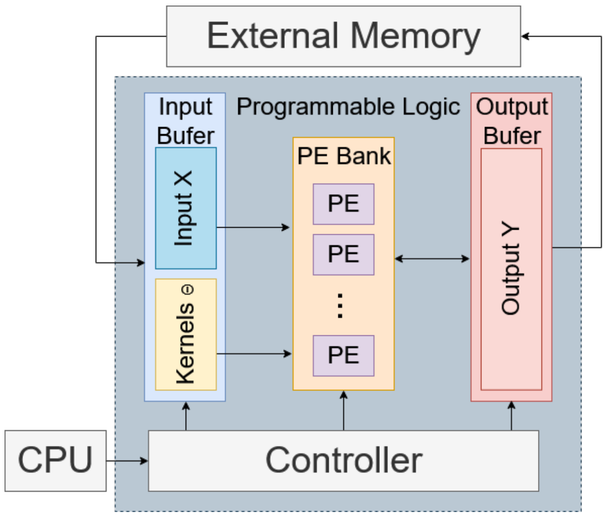


Figure 3.2: SIMD Accelerator, adapted from [19]

3.3.1 Loop Unrolling

In Alg. 1, there are four parallelization opportunities, one for each of the commented loops, each one leading to a different dataflow.

- Loop 4, which iterates over the output channels, exhibits Inter-FM parallelism, meaning that each output FM can be calculated in parallel. For each output FM the same input pixels are used, but a different kernel Θ_n is required. By unrolling loop 4, at each cycle, the same pixel is multiplied by a corresponding weight from different kernels. This leads to an uniform access for input pixels and kernels. All computed values belong to pixels in different output FMs, so all of them need to be stored for the next cycle.

- Loop 3, which iterates over the pixels within the same output FM, presents Intra-FM parallelism by calculating multiple output pixels within the same FM. This process uses the same kernel Θ_n, but different input pixels. Unrolling Loop 3 requires accessing different pixels within the same FM, but enables the reuse of the weights. All the computed values belong to a different output pixel, so the number of partial values that need to be stored corresponds to the unroll factor applied to this loop.

- Loop 2, which iterates over the input channels, highlights Inter-convolution parallelism as each 3D convolution can be described as a sum of multiple 2D convolutions across the input FMs. Each 2D convolution requires different data from the input and kernel. Unrolling Loop 2 leads to an uniform access for input pixels and weights in input FM and 2D kernel coordinates. However each value pair comes from a different channel. All the calculated values can be added into a single partial sum.

- Loop 1, which iterates over a 2D kernel window, reveals Intra-convolution parallelism as each 2D convolution at the two innermost loops can be implemented concurrently. As happens for Loop 2, there is no common data used at each Loop 1 iteration. Unrolling Loop 1 requires accessing different input and weight positions at the same channel. This requires a complex address generation. All the calculated values are used for the same output pixel, therefore can be added, requiring the storage of only one partial sum value.

3.3.2 Loop Tiling

Loop tiling is used if the input data in deep CNNs is too large to fit in the on-chip memory at the same time [19]. Loop tiling divides the data into blocks, like in Fig. 3.1, that are placed in the on-chip memory. The main goal of this technique is to assign the tile size in a way that leverages the data locality of the convolution and minimizes the data transfers from and to external memory. Ideally, each input and weight is only transferred once from external memory to the on-chip buffers.

If the kernel size is bigger than the stride ($J > Stride$ or $K > Stride$), the resulting tiles overlap each other at the boundaries. The added amount of reads is negligible when compared with the tile area [20].

The tiling factors set the lower bound for the size of the on-chip buffer.

3.3.3 Loop Interchange

Loop interchange sets the execution order of the four loops. The innermost loop is the first to be computed and the outermost loop is the last to be completed.

The loop interchange can be divided into intra-tiling and inter-tiling loop order. The intra-tiling loop order determines the dataflow from the on-chip memory to the PEs, since the data belongs to the same tile loaded into the on-chip buffer. While the inter-tiling loop order sets the dataflow from external memory to the on-chip memory, by defining the order of the tiles loaded into the on-chip buffer.

3.3.4 Design Space Analysis

In [20], the loop optimization techniques presented in sections 3.3.1 to 3.3.3 are used to perform an analysis of the SIMD accelerators design space.

The development of the accelerator in [20] is guided by the minimization of the $\underline{\text{computing latency}}$, the requirements of $\underline{\text{partial sum storage}}$ and the accesses to the $\underline{\text{external memory}}$ and $\underline{\text{on-chip buffer}}$.

The computing latency has a lower bound determined by the number of multipliers available. However, if the ratios between the loop tiling factors and unroll factors are not integers, the amount of multiplications done in parallel are less than the number of available multipliers. The same happens if the ratio between the total data size and the tile size is not integer with regards to the external memory transactions. To avoid the underutilization of the resources, the chosen loop tiling factors should be common factors of the total data size, the same needs to happen between the loop tiling and unrolling factors.

The amount of partial sums that need to be stored depends mainly on the order of loop computations. To reduce the number of stored partial sums the output pixel values need to be computed as early as possible. This corresponds to having Loop 1 as the innermost loop.

The accesses to the external memory are tied to the size of the on-chip buffers, which in turn have a size lower bound determined by the loop tiling factors.

The accesses to the on-chip buffers can be reduced by reusing the data sent to the PEs. According to the analysis in 3.3.1, this corresponds to unrolling Loop 3 and Loop 4.

3.3.5 FPGA Implementation

In [20], an accelerator architecture is implemented based on the impact of the loop optimizations in performance.

The accelerator proposed in [20] unrolls loops 3 and 4, which minimizes the on-chip memory accesses.

The tilling is done across W and H of the input data and V and N of the output data. This tiling choice guarantees that the data for Loop 1 and Loop 2 is always buffered and can be computed in sequence. In turn, this reduces the amount of saved partial sums to the product of the unrolling factors. Furthermore, the row tiling at the output requires sequential memory accesses which benefit from DMA data transfers.

Tab. 3.1 presents a comparison between the works in [20–23], which implement the VGG [24] network in FPGA.

The proposed accelerator in [20] achieves over three times the throughput of the other accelerators. The results highlight the impact in computational throughtput of the datapath optimization strategies. For example, between implementations [23] and [20], the same network, at the same frequency achieves over three times the throughput.

Table 3.1: Comparison of CNN FPGA implementations, adapted from [20]

Network [Implementation]	VGG [21]	VGG [22]	VGG [23]	VGG [20]
FPGA	Zynq XC7Z045	Stratix-V GSD8	Virtex-7 VX690t	Arria-10 GX 1150
Frequency (MHz)	150	120	150	150
Number of Weights	50.18 M	138.3 M	138.3 M	138.3 M
DSP Utilization	780 (89%)	1,963[b]	3,600[b]	1,518 (100%)
On-chip RAM[a]	486 (87%)	2,567[b]	1,470[b]	1,900 (70%)
Latency/Image (ms)	224.6	262.9	151.8	47.97
Throughput (GOPS)	136.97	117.8	203.9	645.25
Unrolled Loops	1,2,4	1,2,4	1,2,4	3,4

[a] Xilinx FPGAs in BRAMs (36 Kb) and Altera FPGAs in M20K RAMs (20 Kb)
[b] The resource utilization is not reported in the original papers. The total available resources of the used FPGAs are listed

3.4 CNN Model Optimization

The two most common methods for CNN model optimization in FPGA devices are operand quantization and operation reduction. These methods reduce the amount of resources required, enabling more parallelization and reducing the data transfers.

CNN model optimizations need to take into account the accuracy loss of the model. In order to minimize the accuracy loss, these methods are integrated into the model training phase.

3.4.1 Quantization

Quantization is done by reducing the operand bit size. This restricts the operand resolution, affecting the resolution of the computation result. Furthermore, representing the operands in fixed-point instead of floating-point translates into another reduction in terms of required resources for computation.

The simplest quantization method consists in setting all weights and inputs to the same format across all layers of the network. This is referred as static fixed point (SFP). However, the intermediate values still need to be bit-wider to prevent further accuracy loss.

In deep networks, there is a significant variation of data ranges across the layers. The inputs tend to have larger values at latter layers, while the weights for the same layers are smaller in comparison. The wide range of values makes the SFP approach inviable, since the bit-width needs to expand to accommodate all values.

This problem is addressed by Dynamic Fixed Point (DFP), which consists in the attribution of different scaling factors to the inputs, weights and outputs of each layer.

In [25], a DFP implementation with 8 bits for the weights and 10 bits for the pixel values, without fine-tuning of the weights, is presented. The DFP implementations yielded minimal accuracy losses for the networks tested, as is presented in Tab. 3.2. The fixed-point precision representation leads to an accuracy loss of less than 1%.

Model Accuracy Comparison	**Full Precision**		**Fixed-Point Precision**	
CNN Model	**Top-1**	**Top-5**	**Top-1**	**Top-5**
AlexNet [10]	56.78%	79.72%	55.64%	79.32%
NIN [26]	56.14%	79.32%	55.74%	78.96%

Table 3.2: Accuracy Comparison with the ImageNet dataset, adapted from [25]

3.4.2 CNN Model Reduction

Another way to optimize the CNN model is to reduce the amount of computations necessary for inference. One method to achieve this is through weight pruning.

Weight pruning consists in eliminating or zeroing the lowest weights. After that, the remaining weights are fine-tuned in order to improve the accuracy. Other criteria for pruning can be developed, like energy based approaches. In [27] the energy consumption of each layers is estimated by accounting for the accesses to the multiple memory levels and the energy expended for computation. The amount of energy estimated by a layer determines the order in which the layers are pruned. The more energy demanding layers are pruned first, since the first layers being pruned tend to have better compression ratios than following layers.

Optimization	Dataset	Removed Parameters (%)	Accuracy (%)	Bitwidth
Prunning [28]	Cifar10	89.3	91.53	8 Fixed Point
Prunning [29]	ImageNet	85.0	79.70	32 Float

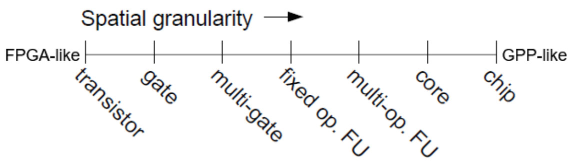
Table 3.3: Accuracy of implementations using CNN model reductions, adapted from [19]

Tab. 3.3 presents results achieved by CNN model reduction techniques. Both implementations manage to remove the majority of the weights without significant impact in accuracy. In fact, [28, 29] claim an accuracy reduction within 1% when compared with the respective model using all parameters.

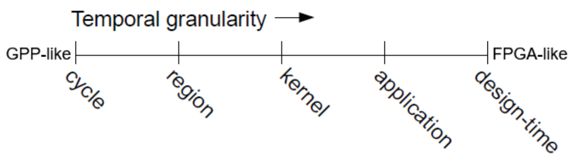
3.5 Coarse Grained Reconfigurable Arrays

Coarse Grained Reconfigurable Arrays (CGRAs) are presented in [30] as middle ground between FPGAs and general purpose processors (GPPs). In fact, a tendency from both FPGAs and GPPs towards CGRAs is highlighted: on the one hand FPGAs tend to have more coarse grained blocks like DSPs, while GPPs become more heterogeneous to include special instructions and accelerators.

In [30], CGRAs are defined as having temporal granularity at the region level or above (Fig. 3.3(b)) and spatial granularity at the fixed functional unit (FU) level or above (Fig. 3.3(a)). As such, CGRAs are able to change the architecture to fit the application during runtime.



(a) Spatial granularity levels



(b) Temporal granularity levels

Figure 3.3: Spatial and temporal granularity levels, adapted from [30]

CGRAs are integrated in systems mainly as accelerators. The runtime reconfigurability allows for changes in the datapath, which allows for hardware reuse in complex applications. At the same time, the FU granularity enables enough degree of control for applications that do not take advantage of bit-level manipulation.

With an adequate tool chain for assembly, compilation and place and route, CGRAs have the potential to facilitate HW acceleration for multiple application types.

Neural networks is one of such applications. All layers in a neural network have the same elemental operations which are mostly MACs and data transfers. Therefore, with tailored FUs and correct datapath configuration, a neural network acceleration environment based on CGRAs seems very attractive.

3.6 Deep Versat

Deep Versat is a CGRA architecture proposed and implemented in [8]. Deep Versat is composed by Versat Layers disposed in a ring structure, as presented in Fig. 3.4.

Each Versat Layer is composed of a Configuration Module (CM) and a Data Engine (DE). This architecture provides a sustainable way to scale the Versat architecture developed in [31]. The scalability comes from the fact that the inputs from a Versat Layer can select data from the same Versat Layer or the previous one. With this connection policy, the routing complexity is independent from the number of deployed Versat Layers.

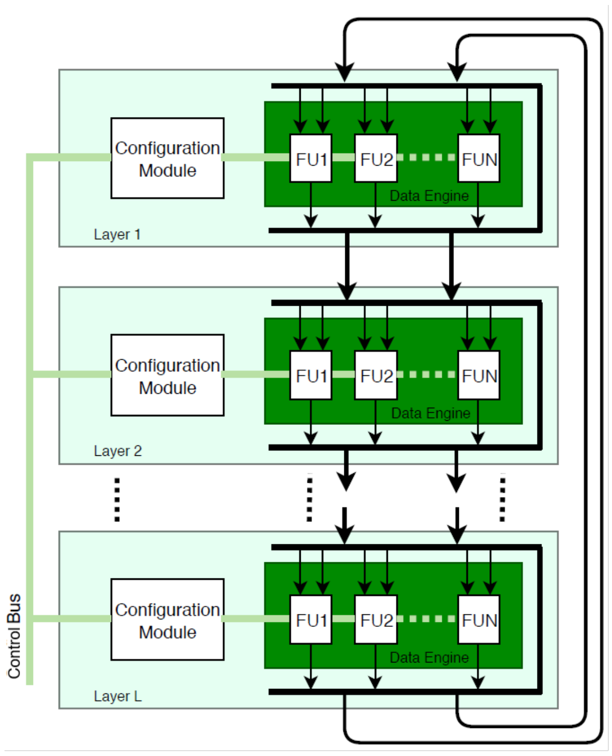


Figure 3.4: Deep Versat architecture, from [8]

3.6.1 Data Engine

The Data Engine (DE) at each Versat Layer is where computations take place. The DE is composed of a set of Functional Units (FUs) in a full-mesh topology. Even though the number of FUs is variable, the topology presents scalability problems, so the author recommends configuring the DE so that each FU input can select one out of a maximum of 20 data sources (10 of then from the previous Versat Layer). A diagram of a DE is presented in Fig. 3.5.

The DE also contains a Configuration Bus to configure each FU with the type of operation and input selection.

The FUs in a typical DE are Arithmetic and Logic Units (ALUs), multipliers, barrel shifters and dual-port embedded memories.

All FUs have a single output port with the exception of the dual-port memories, which have two. The maximum number of FUs is 10, so that the Versat layer does not produce more than 10 data sources that can be selected by the FUs themselves.

The dual-port memories have two inputs and outputs and two Address Generation Units (AGUs) that can generate addresses sequences for the two memory ports. The AGUs are able to generate addresses for variables in nested double loops. Each memory port can read from or write to an address generated by its AGU, an address input to the other port, or can work as a Data Generation Unit (DGU), outputting the sequence generated by the AGU to the memory port itself.

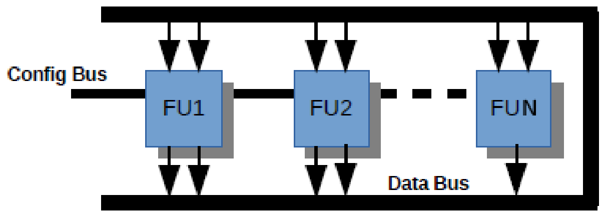


Figure 3.5: Versat data engine, adapted from [8]

3.6.2 Configuration Module

The Configuration Module (CM) is the module where the DE configurations are stored and managed. Fig. 3.6 presents the structure of the configuration module. The main components of this module are the configuration memory, register file and shadow register.

The shadow register holds the configuration currently being executed by the DE. The existence of this register allows for changes in the register file without affecting the DE execution.

The configuration register file is composed of configuration spaces that in turn have multiple configuration fields. One configuration field has the control bits for a particular feature of a single FU, while a configuration space has the set of configuration fields for an FU. Each configuration field can vary in bit width as different the FU features require different numbers of configuration bits. Configuration fields exploit time locality as the whole DE configuration chages little over time. Furthermore, by being addressable at the configuration field level, partial configuration of an FU becomes possible.

The register file takes advantage of time locality, as it is likely that the same DE configuration is valid for a time span or similar configurations are used in sequence.

The configuration memory can store 64 full DE configurations, the most used ones usually, which can be loaded from or uploaded to the register file or external memory, opening the possibility for more configuration storage in the system.

3.6.3 Controller and System Integration

The Deep Versat layers are controlled by a soft processor using a RISC-V architecture. This architecture is programmable with the GNU standard toolchain, which contains compilers for C and C++.

In [8], Deep Versat is connected to the controller as a peripheral using the system control bus. It has a data interface connected to a data bus. Each bus is memory mapped by a distinct memory base address. There are also dataflow buses that connect two consecutive Versat layers.

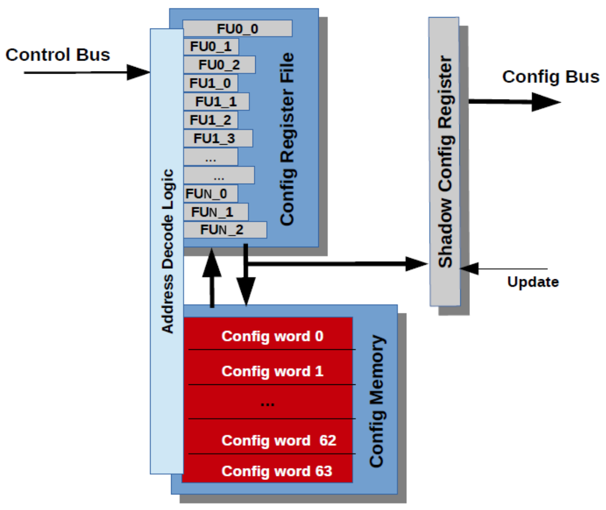


Figure 3.6: Versat configuration module, adapted from [31]

The control bus is used to start a Deep Versat run and verify its completion. Additionally, this bus can access the Versat memory FUs and is also used to manage the configuration registers and memory.

The data bus connects Deep Versat to data sources and sinks and should be used for larger and faster data transfers. The author comments on the inefficiency of driving the data through the RISC-V processor and suggests the implementation of a DMA engine connecting Deep Versat directly to the external memory instead.

Fig. 3.4 presents the system architecture, where there is also a UART peripheral, used mainly for printing debug and verification messages. Each Versat block represents a Versat layer of Deep Versat.

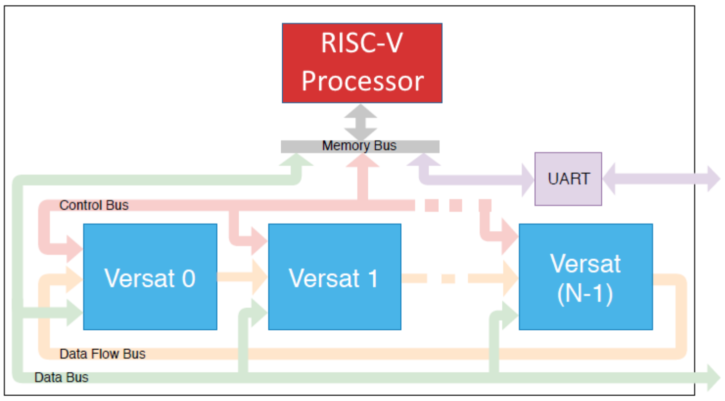


Figure 3.7: Deep Versat system, adapted from [8]

3.6.4 Deep Versat API

To facilitate datapath configuration in Deep Versat, an API in C++ was created where the hardware modules are represented by classes and and their functions and connections are accessed by the class's methods.

First, the definition of the number of Versat layers, types and number of FUs and memory sizes is established. Then, the hardware architectural parameters of Deep Versat are passed into the API via a python script that generates a C++ header file from the Verilog header file containing all the macros used for pre-silicon configuration.

Final Remark

CGRAs have the potential to combine the computational capabilities of FPGAs with the versatility of software, lowering the barrier of entry for model deployment. For that, the CGRA requires dedicated FUs for the CNN layers that exploit datapath and model reduction optimizations, while having an adequate toolchain for software development.

Chapter 4

Proposed Work

The proposed project consists in the development of a Yolov3 CNN application for Deep Versat, with focus on the convolutional layer, that represents the majority of computations in CNNs.

This involves the design of a set of acceleration datapaths for Deep Versat, which, in turn, may require the improvement of the existing FU set. The overall goal is to accelerate the execution of the application up to 30 images per second in order to achieve video speed.

4.1 Yolov3 Software Modeling

The Yolo networks use Darknet [32], an open source neural network in C and CUDA, which does not have support for reduced precision operands. Therefore, the development of a code base that facilitates the evaluation of the Yolov3 network in terms of operand precision becomes a requirement.

This evaluation is necessary to verify the accuracy of the network with reduced precision operands and to measure the ranges of the activations at each layer. Only with this analysis it is possible to implement an effective quantization based on DFP (section 3.4.1), which is necessary for deep networks.

4.2 Baseline Hardware System

In order to establish a baseline for the performance of the Deep Versat, the system presented in Fig. 4.1 will be used. The architecture uses as base the system in [8].

The main difference from the system in [8] is the addition of the DMA connection between the Deep Versat and the external memory. As discussed in 3.6.3, this change frees the host system during data transfers.

A RISC-V processor is used as the host and controller for Deep Versat. This soft processor treats each other block in the system as a memory mapped peripheral. The processor is also tasked to execute the parts of the application that are not accelerated by the Deep Versat.

The UART block is useful in development for debugging purposes, as it is a practical way to receive feedback from the system into a console on an external computer.

Deep Versat has been programmed with a convolutional layer for accelerating a hand written digit recognition application [8]. This implementation uses Versat's current set of FUs, which are reconfigured many times to form datapath configurations as discussed in 3.6.

With this implementation it is possible to both verify the correct functioning of the system and all its components, as well as establishing a baseline in terms of performance for convolution acceleration.

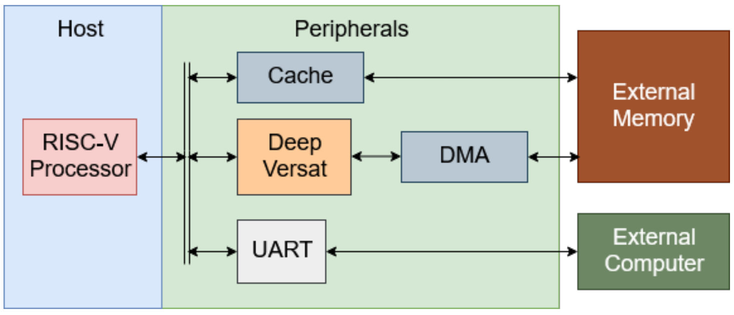


Figure 4.1: Baseline hardware system

4.3 Datapath Proposal for Convolutional Layers

The acceleration of convolutional layers will start with the proposal of a datapath architecture, based on the analysis in 3.3. The accelerator presented in 3.3.5 is a promising starting point for the architecture.

The designed datapaths may require the development of new of FUs or tweaking of the existing ones. There is then the challenge of mapping the convolution datapaths into the Deep Versat architecture using the modified FU set. After that, the convolution layer can be tested and compared with the baseline results. The work reported in this section is the main focus of the project and is expected to be the most time consuming task.

4.4 Experimental Validation with Yolo Networks

With a new Convolutional Layer tested and working properly, the next test is to execute the Tiny-Yolov3 network (section 2.6.3) on the system. This represents an intermediate step due to the reduced requirements of this network, when compared with the full Yolov3. As a general goal for the project, processing 30 images per second would be ideal, although this objective may not be achieved.

4.5 Work Calendarization

In Fig. 4.2 it is presented a GANT chart outlining the calendarization of the planned work.

Month	fev/20				mar/20				abr/20				mai/20				jun/20				jul/20				ago/20				set/20				out/20			
Task/Week	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
SW Baseline			■	■	■																															
Fixed Point Version			▨	▨																																
Value ranges and					▨																															
accuracy analysis					▨																															
HW Baseline						■	■	■																												
Deep Versat w/DMA						▨	▨																													
Test Conv Layer								▨																												
Convolutional Accel.									■	■	■	■	■	■	■																					
Datapath Proposals									▨	▨																										
Map to Deep Versat											▨	▨	▨	▨																						
Tests and Results															▨																					
Yolo Validation																■	■	■	■	■	■															
Program Tiny-Yolov3																▨	▨																			
Tests and Results																		▨																		
Program Yolov3																			▨	▨																
Tests and Results																					▨															
Accelerate other layers																						■	■	■	■											
Datapath Proposals																						▨														
Map to Deep Versat																							▨	▨												
Tests and Results																									▨											
Write Dissertation			■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■				

Figure 4.2: Work Planning

Bibliography

[1] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013.

[2] R. B. Girshick, “Fast R-CNN,” *CoRR*, vol. abs/1504.08083, 2015.

[3] S. Ren, K. He, R. B. Girshick, and J. Sun, “Faster R-CNN: towards real-time object detection with region proposal networks,” *CoRR*, vol. abs/1506.01497, 2015.

[4] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, “Feature pyramid networks for object detection,” *CoRR*, vol. abs/1612.03144, 2016.

[5] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” *CoRR*, vol. abs/1506.02640, 2015.

[6] J. Redmon and A. Farhadi, “Yolo9000: Better, faster, stronger,” *arXiv preprint arXiv:1612.08242*, 2016.

[7] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv*, 2018.

[8] V. Mário, “Deep versat: A deep coarse grain reconfigurable array.” Master’s Thesis, November 2017.

[9] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural Computation*, vol. 1, pp. 541–551, Dec 1989.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[12] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[13] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of IEEE*, vol. 105, pp. 2295 – 2329, December 2017.

[14] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015.

[15] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft COCO: common objects in context,” *CoRR*, vol. abs/1405.0312, 2014.

[16] A. Vidhya, “Yolov3 theory explained.” `https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193`, Visited in 22 Nov 2019, July 2019.

[17] J. Hui, “map (mean average precision) for object detection.” `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`, Visited in 7 Jan 2020, Mar. 2018.

[18] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” *CoRR*, vol. abs/1708.02002, 2017.

[19] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, “Accelerating CNN inference on fpgas: A survey,” *CoRR*, vol. abs/1806.01683, 2018.

[20] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, “Optimizing the convolution operation to accelerate deep neural networks on fpga,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, pp. 1354–1367, July 2018.

[21] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, “Going deeper with embedded fpga platform for convolutional neural network,” pp. 26–35, 02 2016.

[22] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks,” pp. 16–25, 02 2016.

[23] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient cnn implementation on a deeply pipelined fpga cluster:,” pp. 326–331, 08 2016.

[24] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv 1409.1556*, 09 2014.

[25] Yufei Ma, N. Suda, Yu Cao, J. Seo, and S. Vrudhula, “Scalable and modularized rtl compilation of convolutional neural networks onto fpga,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Aug 2016.

[26] M. Lin, Q. Chen, and S. Yan, “Network in network,” 2013.

[27] T.-J. Yang, Y.-H. Chen, and V. Sze, “Designing energy-efficient convolutional neural networks using energy-aware pruning,” pp. 6071–6079, 07 2017.

[28] T. Fujii, S. Sato, H. Nakahara, and M. Motomura, "An fpga realization of a deep convolutional neural network using a threshold neuron pruning," pp. 268–280, 03 2017.

[29] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Hock, Y. Liew, K. Srivatsan, and D. Moss, "Can fpgas beat gpus in accelerating next-generation deep neural networks?," pp. 5–14, 02 2017.

[30] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 235–244, July 2016.

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

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