CGRA-based Deep Neural Network for Object Identification

Pedro José Runa Miranda
pedro.r.miranda@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

January 2021

Abstract

The objective of this work is to accelerate an object detection application using the VersatCNN, a Coarse Grained Reconfigurable Array (CGRA) architecture developed for efficient inference of Convolutional Neural Network (CNN). The performance objective is to enable real-time execution - 30 frames per second - on low-end embedded systems. The object detection application accelerated is the Tiny YOLOv3, based on a CNN developed for constrained environments. The Tiny YOLOv3 network is composed of Convolutional, Maxpool, Route, Upsample and Yolo layers. The application also includes pre and post-processing routines. This thesis work uses the IOb-SoC, a RISC-V based system on a chip (SoC) from IObundle as a baseline hardware platform for the project development. Peripheral modules for measuring performance and enabling ethernet communication are integrated into the IOb-SoC. The embedded version of Tiny YOLOv3 is profilled for RISC-V only execution to identify the parts that require acceleration. The VersatCNN is integraded into the system as a peripheral. The work mainly focuses on the design of CGRA dataflow configurations, as the different parts of the application require distinct configuration strategies for the multiple layer types and the pre and post-processing routines. The final system achieves over 30 FPS for the complete Tiny YOLOv3, with a clock frequency of 143 MHz and a 832x parallelism factor for convolutional operations.

Keywords: Object Detection, Convolutional Neural Networks, Systems on Chip, Coarse Grained Reconfigurable Arrays.

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 into two stages: proposing regions of interest and performing object localization on each one [3, 5]. Other approaches like [11] use an end-to-end solution that makes predictions directly from the input image, trading some accuracy for execution speed. As CNN models evolve, they become increasingly more demanding in terms of computation and memory accesses. Graphical Processing Units (GPUs) normally train and deploy CNNs, since they achieve a high number of operations per second. GPUs leverage time parallelism with Single Instruction Multiple Data (SIMD) architectures, and are efficient for processing multiple images (batches). However, GPUs have large form factors, are expensive and consume significant amounts of energy.

There has been a trend to develop dedicated hardware for accelerating CNN inference using reconfigurable hardware architectures such as Field Programmable Gate Arrays (FPGAs). The intent is not only to increase performance but also to lower power consumption.

In these devices, the power consumption is lower and comes mainly from the memory accesses. Besides, FPGAs allow for the application of a range of optimization techniques. Coarse-Grained Field Arrays (CGRAs) enable the same possibilities for exploiting parallelism as in FPGAs but at a higher level of abstraction. A CGRA is an array of Functional Units (FUs), whereas an FPGA is an array of low-level Look-Up Table (LUT) circuits. The higher level datapath manipulation in CGRAs allows for the creation of toolchains that are simpler for developing accelerators, lowering the barrier of entry for such programmable devices.

The main objective of this work is to develop an implementation of the Tiny YOLOv3 [10] convolutional neural network for an embedded system employing a FPGA or ASIC, capable of achieving a performance of at least 30 frames per second, while detecting and classifying objects in moving pictures or videos. A CGRA architecture geared towards accelerating CNN layers has been developed [9] to achieve the objective, and the Tiny YOLOv3 appli-

cation was modified to run on an embedded environment. This work focuses on the design and implementations of configuration strategies for the VersatCNN that accelerate different CNN layers and other routines.

A SoC platform is used as an hardware baseline for the accelerator deployment. The SoC platform has a RISC-V CPU connected to main memory via a cache system and a peripheral for serial communication. The testing environment requires the integration of an ethernet module to enable large file transfers between a computer and the SoC platform. The SoC platform also requires the addition of a peripheral for performance measurements. The baseline system is used to profile the Tiny YOLOv3 execution and identify target routines to accelerate.

The VersatCNN CGRA is added as a peripheral of the final system. The CPU executes the embedded software and controls the several peripherals, including the VersatCNN configurarions using an C++ Application Programming Interface (API). The CPU also executes parts of the application which have not been hardware accelerated. The different layers of the CNN are mapped into the accelerator core with a set of configuration strategies developed for the embedded software. In addition to the CNN model execution, the application includes functions for pre and post-processing the data. These functions have a significant impact on the global performance of the application. Hence, they also needed to be accelerated.

The work presented uses the IObundle System on Chip (IOb-SoC) [4] platform, which also benefited from other dissertations [2, 13]. Most importantly, the development of the Tiny YOLOv3 network was a significant undertaking, only possible in the scope of two dissertations, this one and [9].

In this work, all system peripherals were integrated within the IOb-SoC platform, including the VersatCNN accelerator. The peripherals integrated are a module for performance measurement and an ethernet communication module. The ethernet comunication also required establishing a communication protocol to transfer the input files and output image between a computer and the embedded device. The VersatCNN accelerator was integrated into the final system. The main contribution of the work consists in the design and implementation of different configuration strategies for the execution of the several Tiny YOLOv3 routines and CNN layers in the VersatCNN.

2. Background

2.1. Convolutional Neural Networks (CNNs)

CNNs are mainly composed of Convolutional layers. Other commonly found layers in CNNs are Pooling,Batch-normalize, Routing and Upsample layers.

The input of Convolutional layers is divided into C channels, where each channel is defined as a $(W \times H)$ feature map (FM). Each kernel used in the convolution has C channels. The number of output channels N 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.

CNNs use Pooling layers to reduce the size of the FMs. 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.

Batch-normalize layers set the average of the input values to zero and the standard deviation 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. For a given value y, the normalization outputs

$$z = \frac{y - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta, \tag{1}$$

where μ is the average, σ is the standard variation, ϵ the scale and β is the batch normalization bias. The additional ϵ term is used to avoid numerical errors related with denominators too close to zero.

Routing layers concatenate the output of all the selected layers.

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

2.2. Tiny YOLOv3

The Tiny YOLOv3 is a small version of the YOLOv3 [11] CNN, for constrained environments.

The first part of the network is composed of a series of Convolutional and Maxpool layers. Maxpool layers reduce the FMs by a factor of four along the way.

The object detection and classification part of the network performs object detection and classification at (13×13) and (26×26) grid scales.

The detection at a lower resolution is obtained by passing the feature extraction output over 3×3 and 1×1 Convolutional layers and a Yolo layer at the end.

The detection at the higher resolution follows the same procedure but using FMs from two layers of the network. The second detection uses intermediate results from the feature extraction layers concatenated with upscaled FMs used for the lower resolution detection.

2.3. CNN Inference Acceleration in FGPAs

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

The convolutional algorithm has four options for loop unrolling: inter-FM parallelism to compute each output FM (OFM) in parallel; intra-FM parallelism to compute multiple output pixels within the same OFM in parallel; inter-convolution parallelism to compute MAC operations of different IFMs in parallel; and intra-convolution parallelism to compute MAC operations of the same 2D kernel window in parallel.

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. Loop tiling divides the data into blocks 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.

The CNN model optimization in FPGA devices includes operand quantization. This methods reduce the number of resources required, enabling more parallelization and reducing data transfers.

Quantization is done by reducing the operand bit size. This restricts the operand resolution, affecting the resolution of the computation result. In Dynamic Fixed-Point (DFP) quantization, the inputs, weights and outputs of each layer have different scaling factors.

2.4. Coarse Grained Reconfigurable Arrays

Coarse-Grained Reconfigurable Arrays (CGRAs) are presented as a middle ground between FPGAs and general-purpose processors (GPPs). In [15], CGRAs are defined as having temporal granularity at the region or loop-nest level or above and spatial granularity at the fixed functional unit (FU) level or above . As such, CGRAs can change the architecture to fit the application during runtime.

CGRAs are integrated into 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.

Deep Versat is a CGRA architecture proposed and implemented in [7]. Deep Versat is composed of layer of fully-connected FUs organized in a ring structure.

3. System Baseline

3.1. Embedded Software Version

The execution of the Tiny YOLOv3 application in an embedded environment requires changing all memory allocations to static memory regions predefined at compile time due to lack of operating system. The data format is also changed from floating-point to 16-bit fixed point.

The CNN model execution also benefits from performance optimizations such as batch-normalize folding and linear approximations of the activation functions. The model accuracy is also improved by implementing dynamic fixed-point post-training quantization.

The batch-normalize folding combines the convolutional weights w_{cjk} and biases b with the scale γ, bias β, average μ, standard deviation σ and ϵ parameters from batch-normalize layer into a new convolutional bias b' and set of weights w'_{cjk} given respectively by

$$b' = \frac{b-\mu}{\sqrt{\sigma^2-\epsilon}}\gamma + \beta, \texttt{and } w'_{cjk} = \frac{w_{cjk}}{\sqrt{\sigma^2-\epsilon}}\gamma. \quad (2)$$

Both leaky and sigmoid activations are approximated by linear functions which require less resources to implement in hardware. The leaky activation coeficient α can be approximated by

$$\alpha' = 2^{-4} + 2^{-5} + 2^{-7} = 0.1015625. \quad (3)$$

The sigmoid activation is approximated by a piecewise linear function proposed in [14].

The implemented dynamic fixed-point model allows for independent fixed-point formats for the activation, weights and biases in each layer. The weight and bias formats are determined from the original floating-point weight file. The number of integer bits is the minimum that accomodates for the detected value range. The output FM format is determined from the value range detected during the inference of the COCO [6] test dataset.

All the neural network models perform inference over the MS COCO 2017 test dataset [6]. The final model uses 16-bit fixed-point weights and biases with the previously explained approximations and optimization achieves a 30.8 mAP_{50}, versus the 32.9 mAP_{50} of the original model.

3.2. IOb-SoC Hardware Platform

The baseline hardware system used for this work is IOb-SoC [4], an open-source RISC-V SoC platform developed by IObundle. Figure 1 presents the block diagram of the system. A RISC-V soft CPU core controls the system.

The CPU used is the PicoRV32 RISC-V core [16], minimally modified to be integrated into IOb-SoC. The PicoRV32 CPU is designed to use minimal hardware resources and, as a consequence, has very

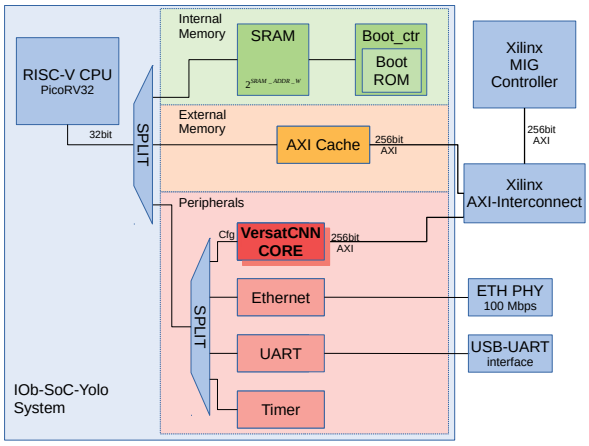

Figure 1: IOb-SoC-Yolo block diagram.

low performance, taking 4 Cycles Per Instruction (CPI). The CPU can access the internal memory, the external DDR memory (via cache), and the four peripherals mentioned:

- the VersatCNN Accelerator Core is the main focus of this work and is used to accelerate the convolution operation.
- the Timer is used to measure performance.
- the UART is used for programming and basic user runtime messages.
- the Ethernet module provides a higher bandwidth communication facility used to transfer large datasets to IOb-SoC.

3.3. Tiny YOLOv3 Profiling

Table 1 presents the execution times for the Tiny YOLOv3 application on the presented IOb-SoC with a frequency of 143MHz. As expected, the CNN is resposible for the majority of the computation, taking 99.9% of the total execution time.

The profiling also includes the CNN pre and post-processing functions of the application. The CNN pre-processing resizes the input image from size $Img_W \times Img_H \times Img_C$ to size $416 \times 416 \times 3$, which is the required size for processing. The CNN post-processing searches the CNN output for scores over a threshold value, applies NMS to the potential detections and draws the detections into the input image. In addition to the CNN, both processes require acceleration to achieve the target performance.

4. VersatCNN CGRA

The versatCNN is the result of multiple architectural improvements to the Deep Versat CGRA developed in [7]. Figure 2 presents the detailed architecture of the VersatCNN Core.

Tiny YOLOv3 Part	Execution Time (s)
Pre-CNN	1.040
CNN	967.746
Post-CNN	0.014
Total	968.800

Table 1: Tiny YOLOv3 RISC-V-only performance.

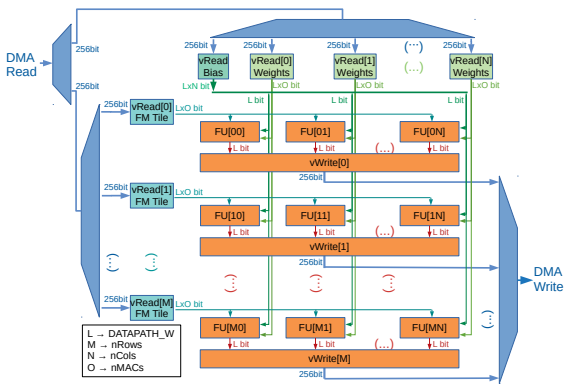

Figure 2: VersatCNN detailed block diagram.

The VersatCNN uses an AXI DMA (Direct Memory Access) to read and write data from the main memory. The DMA has 256-bit data width and with AXI bursts the bandwidth averages to almost one 256-bit word per cycle for both reads and writes.

The data from main memory is stored in buffer memories called vReads. There are two sets of vRead FUs in the architecture: one for IFMs and another for weight kernels and biases. The vRead FUs send the data into the custom computational FUs.

The custom FUs are organized in a two-dimensional matrix to enable intra-FM parallelism in the same column and inter-FM parallelism in the same row.

The results from each custom FU line are concatenated and written into an output memory buffers called vWrite. The data from the vWrite is transfered back into main memory through the DMA.

The VersatCNN dataflow is divided into three phases: *memory read*, *compute* and *memory write*.

In the *memory read* phase, the data is transferred from the main memory to the vRead FUs. In this phase, the read operations from the main memory and the writes to the vRead FUs are controlled external address generator units (AGUs).

The *compute* phase sends the data from the vRead FUs for computation in the custom FUs and writes the result in the vWrite FUs. In this phase,

all the internal AGUs control the reads from the vRead FUs, the operations at the custom FUs and the writes to the vWrite FUs.

In the *memory write* part the data from the vWrite FUs is transferred back to main memory. A external AGU controls the read accesses to the vWrite FUs and the write transfers to main memory.

This architecture allows the pipelining of consecutive dataflows. The execution of a second dataflow starts after the *memory read* phase of the first dataflow is completed. At that point, the CGRA executes the *memory read* phase of the second dataflow and the *compute* phase of the first dataflow simultaneously.

The number of vRead FUs for kernels used in the design is defined by the `nCols` parameter. The number of vRead FUs used for the input feature maps is defined by the `nRows` parameter. With this design, the number of vRead FUs varies with `nCols` + `nRows` and the number of vWrite FUs with `nRows`, while the number of custom FUs for computation is `nCols` × `nRows`. The `nMACs` parameter defines the number of MACs in each custom FU.

4.1. Address Generator

The vRead and vWrite FUs, as well as the execution of the custom FUs, are controlled by AGUs. The versatCNN uses a total of nine groups of AGUs. The buffer memories are divided into four groups: vReads for weights, a single vRead for biases, vReads for input FMs and vWrites for OFMs. Each group of memories is controlled by a pair of AGUs: one for the external accesses and another for the internal accesses. All the custom FUs for computation use a single AGU to generate control signals.

An AGU can access the memories in a nested loop pattern, which is described by algorithm 1.

```
addr = start
for i ∈ {1, ..., iterations} do  // Outer loop
  for p ∈ {1, ..., period} do  // Inner loop
    addr += incr
  addr += shift
```

Algorithm 1: Address generator pattern.

The inputs `start`, `iterations`, `period`, `incr` and `shift` determine the pattern outputted in the `addr` port.

A pattern with more nested loops is achieved by chaining multiple AGUs. Each additional chained address generator unit adds a set of `iter`, `period`, `shift` and `incr` configurations.

4.2. Custom Functional Unit

The custom FU for computation receives as input the feature map values, biases and weights from the vRead FUs. The custom FU outputs the computation result to a vWrite FU.

The custom FU for computation enables inter-convolution parallelism, where multiple IFM channels are computed in parallel using multiple MACs. This parallelism requires multiple words of IFM data and kernel weights.

The number of input feature map channels computed in parallel is parameterizable by the `nMACs` parameter.

The custom FU can be configured to perform multiple operations: MAC output, convolution, convolution and maxpool, maxpool and data bypass.

The convolutional operation in the custom FU can be further configured with regards to the activation function: leaky, sigmoid or linear. For each configured operation the bias can be used or not.

5. Tiny YOLOv3 with VersatCNN

The Tiny YOLOv3 application receives an input image of dimentions $\texttt{IMG}_\texttt{W} \times \texttt{IMG}_\texttt{H}$. The input image is resized to 416×416 resolution in the CNN pre-processing part. The resized image is the CNN input. The results from the CNN are post-processed in three main steps. The application checks for detections with objectness score over the pre-defined threshold. The next step filters overlapping boxes using non-maximum suppression. The filtered boxes and respective classes are drawn in the original input image.

The profiling done in section 3.3 shows that all parts of the Tiny YOLOv3 application require acceleration to meet the real-time (30 FPS) objective. The final program for the application also includes functions for initial setup and system configuration. These functions include initial peripheral configuration, resetting the main memory and the transfer of the weight file and input image into the main memory. The initial setup also includes functions to calculate auxiliar arrays for pre and post-CNN processing.

5.1. Data Storage Format

The data storage format implemented in the embedded software influences the loops for convolution. To reduce the number of loops for data access during convolution, the data is stored in memory in a ZXY format instead of the original XYZ format. Figure 3 presents a scheme of both formats for a $4 \times 4 \times 3$ input feature map.

The XYZ format stores the values by column and row of each feature map. In ZXY format, the data is stored by channel first. Figure 3 also highlights the values used to compute the first pixel of a convolution output. The XYZ format disperses the input values in 9 groups of 3 contiguous values. This format requires 3 loops to iterate across all values: one

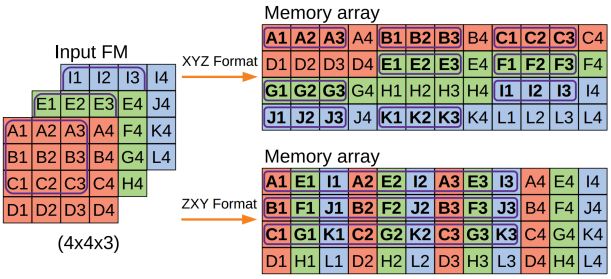

Figure 3: Data storage formats.

loop to iterate across the values in each FM row, one loop to iterate across the rows of an FM and one to iterate across different FMs. The ZXY format stores the input values in 3 groups of 9 contiguous values each. This format requires 2 loops to access all values: one loop to iterate across all rows and one loop to iterate, in each row, across all values and channels.

5.2. CNN Acceleration

As discussed in section 3.3, all types of layers require acceleration. This section presents the different strategies implemented for acceleration.

5.3. Convolution with Maxpool

The first 8 layers of the network can be executed in Convolutional+Maxpool layer pairs. To minimize data transfers, each IFM tile is read from main memory once and the corresponding OFM tile is computed for all kernels. With this strategy, all the kernels are loaded from memory for the same set of IFM tiles. For the first four Convolutional layers, the kernels are small enough to fit into the vRead memory buffers at the same time, therefore the kernels are only loaded once from main memory.

Most configurations stay the same across the complete Convolutional and Maxpool layers execution. The variable configurations are mostly related to updating the data pointers for reading and writing data to/from main memory.

The *Memory Read* phase configurations determine the dataflow of the input data from the main memory to the vRead buffers.

The VersatCNN is configurated to transfer a set of `nCols` kernels and biases from main memory. The set of kernels and biases are transferred in the same DMA burst, as they are stored sequentially in main memory.

Each kernel and corresponding bias amount to $\texttt{ker\_side} \times \texttt{ker\_side} \times \texttt{C} + 1$ values. Each transfer reads 16 values from memory (each value is 16-bit and $256 = 16 \times 16$).

The pointers to main memory and to the vReads for the kernels and biases are updated for each set read from memory. After the first set of IFM tiles is processed, all kernels and biases are loaded into the vRead buffers, therefore the weights and biases reads from main memory are disabled.

The size limitations of the on-chip memory buffers require the use of tiling for the IFMs. Figure 4 presents a diagram of the tiling strategy used for the application.

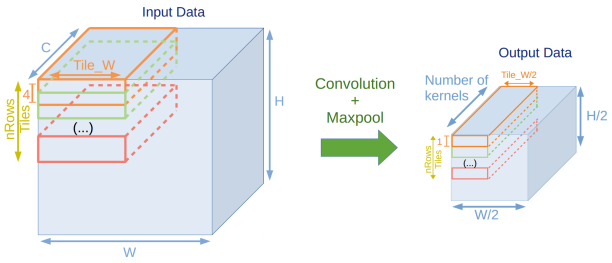

Figure 4: Data tiling diagram.

The tiling implemented divides the IFM into blocks of 4 rows, $\texttt{Tile\_W}+2$ columns and `C` channels. The two extra columns per tile are used as padding to maintain the pattern for convolution on the tile edges. Since there are `nRows` vReads to store IFM tiles, the same amount of IFM tiles are transferred from main memory.

Each IFM tile starts two rows after the start of the previous tile. Note that an IFM tile transfer requires $\texttt{ker\_side} + 1 = 4$ bursts, one for each tile row. After each DMA burst, the the pointer to main memory is updated from the end of the tile row to the start of the next tile row. The vRead for IFM tiles operates in ping-pong fashion: the half of the vRead memory used to write data from main memory toggles between two consecutive datapath phase executions.

The *Compute* phase configurations set the dataflow from the input data in the vRead buffers, across the custom FU for computation and into the vWrite buffer that stores the results.

The read ports on the weights vReads output `nMACs` weights in parallel, while the bias vRead outputs `nCols` values simultaneously.

The vRead for weights is read in sequence, taking $\texttt{ker\_side} \times \texttt{ker\_side} \times \texttt{C}/\texttt{nMACs}$ operations to read a kernel. The kernels are read $2 \times \texttt{Tile\_W}$ times, one per each convolution done over an IFM tile. After each convolution, the internal AGU points back to the initial address.

The IFM tile configurations for the *Compute* phase are constant since the accelerator only operates over one set of IFM tiles at a time. The internal AGU that controls the vRead accesses is configured to generate a five-loop pattern.

The two inner-most loops generate an access pattern to perform a single convolution. The third and forth loops generate the pattern for the 2×2 maxpool. The fifth loop generates the pattern to move

accross the `Tile_W`/2 output 2×2 squares of the IFM tile.

Like the vReads for weights, the vReads for IFM values output `nMACs` values in the same word.

The custom FU takes `ker_side` $\times$ `ker_side` $\times$ `C`/`nMACs` iterations for each convolution, for a total of $2 \times$ `Tile_W` convolutions per IFM tile.

The vWrite buffers are written once for every four convolutions. The internal AGU starts the address generation after a delay that accounts for the read process from the vRead memories and the custom FU latency.

Each vWrite receives `nCols` values in parallel, one from each custom FU in the associated row (check figure 2). The results from each set of kernels are written in a pattern that maintains the ZXY format for the output data.

The *Memory Write* configurations set the dataflow to write the OFM tiles into main memory.

The DMA is configured with the the number of bytes per burst. Each burst is sent with an offset of one OFM row ($2($`W/2+2` $\times$ `n_kernels` bytes).

The OFM tile is only written after all sets of kernels are computed.

5.4. Other Convolutional Layers

The remaining Convolutional layers of Tiny YOLOv3 are accelerated in isolation (only Convolutional layer) or combined with a Yolo or Upsample layer.

After layer 8, the IFMs are either 26×26 or 13×13 and therefore there is no tiling across columns. The data for the `nRows` IFM tiles read from main memory is in a block of sequential data. Therefore, the data transfer can be configured as a single burst.

The Convolutional layers 16 and 23 are paired and executed with the Yolo layers 17 and 24 respectively in the same datapath configuration. The Convolutional layers 16 and 23 have linear activation and the Yolo layer operations are equivalent to performing sigmoid activation to selected input channels. Combining both layers consists of performing a regular Convolutional layer with sigmoid activation.

The Convolutional layer 19 is paired with the following Upsample layer. The vWrite internal AGU is configured to store the convolution each convolutional result twice, which automatically performs the width upsampling. The height upsampling is done by configuring the vWrite external AGU to write each OFM tile twice to main memory.

5.5. Maxpool Layer

The Maxpool layers 10 and 12 are accelerated in isolation. The first dataflow reads the complete layer input into the IFM tile vReads. Each vRead has a $2 \times$ `W` $\times$ `C` IFM tile. Each dataflow performs maxpooling over a $2 \times 2 \times$ `C` input block and outputs a $1 \times 1 \times$ `C` output block. Each output block is written to main memory at the end of the dataflow configuration. Each VersatCNN row performs the maxpooling of a 2×2 region at a time, therefore there is a parallelism of `nRows` for the maxpool acceleration.

5.6. Pre-CNN Processing

The CNN pre-processing consists in resizing the input image by performing two linear approximations over the image: one for width resizing and another for height resizing.

The access patterns cannot be mapped into an AGU configuration. Therefore the pattern is calculated by the CPU before the application execution. During the pre-CNN process, the pattern is loaded into a memory used to store patterns for accessing the IFM tile vReads, enabling the acceleration of the width and height resize loops.

For the width resize loop, each IFM vRead receives an input image row and the weight vReads receive the resize factors. Each custom FU row calculates all image channels in parallel, for a total parallelism factor of `nRows` $\times$ 4, where one of the channels is used for padding.

In the width resize loop, only the first row is used for computation. The first row computes all image channels in parallel.

5.7. Post-CNN Processing

The CNN post-processing part of the application is composed of three different functions. The functions to create and filter boxes have a workflow dependent on control instructions which are not suited for acceleration using the VersatCNN CGRA. The function to draw detections presents a regular workflow that can be mapped into the VersatCNN.

The drawing of each detection can be accelerated in two routines: one for drawing the class label and another to draw the detection box around the object.

The drawing of the class labels is accelerated in VersatCNN by loading the class label image into the IFM tile vReads and the respective class colour into the weights vReads. The final result is written directly over the input image. The process is repeated for each row of the class label image.

The process for drawing the detection box requires writing the correct class colour into the input images. The process is divided into two distinct sets of configurations. The first configuration writes the horizontal sides of the detection box. The vertical sides require two write operations, one for each box side, for the rows that make up the box height.

6. Results

The IOb-SoC-Yolo system uses the IOb-SoC platform with the VersatCNN CGRA to accelerate the Tiny YOLOv3 application. Table 2 lists the parameters for the final VersatCNN version used.

Parameter	Value
nCols	16
nRows	13
nMACs	4
DDR_ADDR_W	32
DATAPATH_W	16
VREAD_BIAS_ADDR_W	3
VREAD_WEIGHT_ADDR_W	14
VREAD_TILE_ADDR_W	15
VREAD_PATTERN_ADDR_W	10
VREAD_TILE_EXT_ADDR_W	15
VWRITE_ADDR_W	8

Table 2: VersatCNN parameters for the IOb-SoC-Yolo implementation.

With the chosen paramteres, the VersatCNN is capable of up to $16 \times 13 \times 4 = 832$ parallel MAC operations.

The system is synthesized for a Xilinx XCKU040 FPGA with a clock frequency of 143 MHz.

6.1. Performance Results

Table 3 presents the Tiny YOLOv3 network execution times on multiple platforms. For each platform, both the CNN function and total application execution times are measured. The total application includes pre and post-processing in addition to the CNN function.

The CPU and GPU results have been obtained using the original Tiny YOLOv3 network [12]. The FPGA results use the embedded software version discussed in section **??**.

Platform	CNN (ms)	Total (ms)
CPU (i7-8700)	819.2	828.3
GPU (RTX 2080ti)	7.5	15.4
IOb-SoC	967 745.6	968 799.5
IOb-SoC-Yolo	**24.4**	**30.9**

Table 3: Tiny YOLOv3 execution times on multiple platforms.

The Tiny YOLOv3 on desktop CPUs is too slow for real-time inference. The inference time on an RTX 2080ti GPU shows a 109 speedup versus desktop CPU. On the CPU, the CNN kernel takes about 99% of the total execution time. With GPU acceleration, the CNN kernel execution time drops to under 50% of the total execution time.

Running the complete application on the IOb-SoC-Yolo system, using only the RISC-V CPU for computation takes over 16 minutes. Inference takes over 99.8% of the total execution time, as discussed in section 3.3. Like the desktop CPU application, the CNN is responsible for most of the processing time.

Using the VersatCNN accelerator, the inference time drops to 24.4 ms, which represents 2.26% of the total execution time. Performing inference with the VersatCNN is 39731 times faster than using the RISC-V CPU alone. Since the computational parallelism factor is only 832 (number of parallel MAC units), the remaining speedup obtained is explained with the increased data bandwidth.

With hardware acceleration, both the CNN pre and post-processing times are reduced. The acceleration of both parts only uses one line of FUs with a parallelism factor of `nYOLOmacs` $= 4$. Therefore, most of the pre-CNN speedup comes from the improved memory bandwidth of using the DMA instead of the RISC-V cache system. The post-CNN acceleration is less pronounced as only one of the four post-CNN functions is accelerated. Furthermore, small bursts write the detection boxes and labels to memory which reduces the efficiency of the DMA transfers.

With the acceleration of CNN pre and post-processing, IOb-SoC-Yolo achieves a real-time performance of over 30 FPS, as shown in table 3.

6.2. Resource Utilization

Table 4 presents FPGA resource utilization of the final system for a Xilinx XCKU040 FPGA device of the Kintex UltraScale product family. The IOb-SoC-Yolo column indicates the resources used for the implementation of the IOb-SoC platform, peripherals and VersatCNN CGRA.

The design utilizes about 50% of the available Look-Up Tables (LUT) logic, 18% of the LUT memories, 45% of the Flip-flops (FF) and 45% of the Digital Signal Processors (DSP). The design uses about 65% of the available 36 kB Block RAMs (BRAM).

The majority of the resources is used to implement the VersatCNN CGRA. The VersatCNN uses 832 DSPs for the custom FU MAC blocks. The remaining DSPs are used for the calculation of the external address offsets for ($\texttt{nCols} - 1 = 15$) weight vReads, ($\texttt{nRows} - 1$) IFM tile vReads and ($\texttt{nRows} - 1$) vWrites.

6.3. Comparison With Other FPGA Implementations

The only competing Tiny YOLOv3 FPGA implementations found are [18], [1] and [8]. Table 5 compares IOb-SoC-Yolo with these works in terms of performance.

The implementation in [18] uses a Zynq 7020

Resource	VersatCNN	IOb-SoC-Yolo
LUT logic	103655	106892
LUT memory	16792	17120
FFs	86319	88460
36kB BRAMs	339.0	356.0
DSPs	871	875

Table 4: IOb-SoC-Yolo resource utilization in a Xilinx XCKU040 FPGA.

FPGA with significantly fewer resources than [1] and this work. The FPGA is included in a Zedboard device that uses a dual-core ARM CPU that runs at 667 MHz, while [1] and this work use soft CPU cores running at the same frequency as the developed hardware accelerators.

The number of DSP blocks used in [18] is about 7× lower than those used in IOb-SoC-Yolo, while [1] uses over 2.5× more DSP blocks than IOb-SoC-Yolo, which shows the number of DSP blocks both for the complete system and the inference accelerator.

The MMAC/s figure measures the number of multiply+accumulate operations performed per second by each of the works. For [18] and this work, the MMAC/s figure is obtained by

$$\texttt{MMAC/s} = \frac{\texttt{\#CNN MACs}}{\texttt{CNN Execution Time (s)}}. \quad (4)$$

The Tiny YOLOv3 network has a total workload of 2.78 GMAC operations. Only the Convolutional layer MACs are accounted to calculate the workload. The GMAC/s figure reported in [1] is obtained by

$$\texttt{GMAC/s} = \texttt{\#DSP} \times \texttt{Freq}. \quad (5)$$

This value represents the maximum achievable throughput of the design. The real throughput of the [1] implementation is expected to be lower.

The DSP efficiency figure is presented to account for the different scale of resources used in the different implementations. This work presents over 4× more DSP efficiency than [18]. Even when compared with the maximum achievable throughput in [1], this work obtains a higher DSP efficiency. This is expected as [18] and [1] only perform acceleration for the Convolutional layers, while this work accelerates all network layers.

In [8], is claimed a performance of over 100 FPS using a software and hardware co-design approach. The Convolutional, Maxpool and Upsample layers are accelerated in programmable logic. The resource usage and the clock frequency of the final system is unknown. The pre and post-processing parts of the application are executed in the processing system of the device.

Notice that this performance analysis encompasses only the neural network execution. The CNN pre and post-processing parts of the Tiny YOLOv3 application are outside the scope of both the other works. Although the CNN part is responsible for most of the computational load, the pre and post-processing parts become the bottleneck for real-time execution, as shown in section 3.3.

7. Conclusions

This work describes the development of IOb-SoC-Yolo, a RISC-V SoC with a CGRA accelerator that enables real-time (30 FPS) execution of the Tiny YOLOv3 neural network. In particular, focuses on the design of configurations strategies for the VersatCNN CGRA to enable acceleration of the CNN layers and other routines.

The software model is drastically modified from the original, to suit embedded execution. The final neural network model scores 30.8 $\texttt{mAP}_{50}$ in the COCO 2017 test dataset, while the original model scores 32.9 $\texttt{mAP}_{50}$.

The IOb-SoC platform uses a RISC-V soft CPU core that has access to static on-chip memory and DDR external memory. The system also has a UART peripheral used to load the application firmware and print debug messages on the user's host PC. To establish an effective development and testing environment, this work integrated additional timer and ethernet modules as system peripherals.

The VersatCNN CGRA is designed for a *memory read* → *compute* → *memory write* workflow. The use of a DMA module increases the data bandwidth by bypassing the CPU + cache system. A matrix of custom computational FUs massively accelerates the computation.

A set of configuration strategies for the embedded software is developed to accelerate the application. The configurations include strategies to accelerate Convolutional layers or Convolutional layers paired with Maxpool, Yolo or Upsample layers. Beyond the CNN inference, the image resizing and the drawing of the final detection boxes and labels are also accelerated to achieve the target performance.

The final system achieves a performance of over 30 FPS to execute the complete Tiny YOLOv3 application. To the best of the author's knowledge, this is the only work that attempts to accelerate the CNN execution by improving all layer types. This work is also the only known work that provides acceleration solutions for the complete Tiny YOLOv3 application beyond CNN inference.

The acceleration of the non-convolutional parts of the application highlights the relevance of the data bandwidth in the final performance. In those cases, the achieved speedup is, at times, orders of mag-

Table 5: Performance comparison with other FPGA implementations.

	[8]	[18]	[1]	IOb-SoC-Yolo
Device	Zynq UltraScale+ XCZU9EG	Zynq 7020	Virtex-7 XC7VX485T	UltraScale XCKU040
HW Freq. (MHz)	-	100	200	143
CPU Freq. (MHz)	1.5GHz[a]	667	200	143
DSPs (CNN only)	-	120	2304	878 (832)
GMAC/s	290.3	5.2	230.4[b]	114.2
MMAC/s/DSP	-	33	100	137
CNN Exec. (ms)	9.6	532.0	-	24.4
Quantization	8-bit	16-bit	18-bit	16-bit

[a] Maximum clock for the device [17].
[b] Peak performance.

nitude higher than the parallelism factor exploited for computation.

References

[1] A. Ahmad, M. A. Pasha, and G. J. Raza. Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[2] A. Charana. Development Environment for a RISC-V Processor. Master's Thesis, July 2020.

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

[4] I. Lda. IOb-SoC. `https://github.com/IObundle/iob-soc`, 2020.

[5] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie. Feature Pyramid Networks for Object Detection. *CoRR*, abs/1612.03144, 2016.

[6] 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*, abs/1405.0312, 2014.

[7] V. Mrio. Deep Versat: A Deep Coarse Grain Reconfigurable Array. Master's thesis, Instituto Superior Tcnico, November 2019. Master's Thesis.

[8] S. Oh, J. H. You, and Y. K. Kim. Implementation of Compressed YOLOv3-tiny on FPGA-SoC. In *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, pages 1–4, 2020.

[9] D. Pestana. Object Detection and Classification on the Versat Reconfigurable Processor. Master's Thesis, Jan 2021.

[10] J. Redmon. YOLO: Real-Time Object Detection. `http://pjreddie.com/darknet/yolo/`, visited in 20 Nov 2020, 2018.

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

[12] J. Redmond. darknet. `https://github.com/pjreddie/darknet`, 2018.

[13] J. Roque. Development Environment for a RISC-V Processor: Cache. Master's Thesis, Jan 2021.

[14] I. Tsmots, O. Skorokhoda, and V. Rabyk. Hardware Implementation of Sigmoid Activation Functions using FPGA. In *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, pages 34–38, 2019.

[15] 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)*, pages 235–244, July 2016.

[16] C. Wolf and et. al. PicoRV32 - A Size-Optimized RISC-V CPU. `https://github.com/cliffordwolf/picorv32`, 2019.

[17] Xilinx. *Zynq UltraScale+ MPSoC Data Sheet v1.8*. Xilinx, October 2019.

[18] Z. Yu and C. Bouganis. *A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny*, pages 330–344. 03 2020.