Object Detection and Classification on the Versat Reconfigurable Processor

Daniel Garigali Pestana
daniel.pestana@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

January 2021

Abstract

The main goal of this work is the development of VersatCNN, an IP core based on the Versat Coarse-Grained Reconfigurable Array, extended to efficiently compute Convolutional Neural Networks (CNNs). VersatCNN is validated with the deployment of a state-of-art object detector. VersatCNN is composed of a large number of Multiply-Accumulate (MAC) units embedded in vector units, organised in a matrix structure to exploit parallelism at the convolution and feature map levels and to enhance data sharing. Parallel memory read and write units exchange data with the external memory over a wide memory controller bus. The reconfigurable computing units form different datapaths for accelerating different CNN layers and activation functions. The state-of-art object detector used is YOLOv3-Tiny, a lightweight version of the YOLOv3 detector targeting embedded systems, which has the best trade-off between accuracy and execution time. In this work, the source code is converted to fixed-point and optimised for hardware acceleration using approximated activation functions, batch-normalization folding and post-training dynamic quantization. The precision drop is only 2.1 using the Mean Average Precision (mAP) metric, when compared to the original floating-point model. The YOLOv3-Tiny detector, running the optimised software on a minimal and low performance RISC-V CPU and using the VersatCNN IP core for acceleration, is prototyped in a UltraScale XCKU040 FPGA and achieves a performance of 32.4 frames per second, running at 143 MHz for 768x576 sized images and a parallelism factor of 832 (number of MAC units).

Keywords: Coarse Grained Reconfigurable Array, Convolutional Neural Networks, Versat Reconfigurable Processor, YOLOv3-Tiny, RISC-V CPU

1. Introduction

Object detectors have a wide range of application fields such as security, transportation, military and medical. Their task consists in classifying and locating multiple objects in an image from predefined categories. Object detection has been under extensive research in both academia [4, 5] and real world applications [14, 11]. Traditional approaches were based on handcrafted low-level features and shallow trainable architectures.

Recent technological breakthroughs led to the fast evolution of object detectors. The main contributions include the development of Deep Neural Networks (DNNs) and the increase of the hardware computing power. State-of-art object detectors use DNNs with deeper architectures to learn more complex features without the need to design them manually. The superior accuracy of DNNs comes at the cost of high computational complexity. Graphics Processing Units (GPUs) have been the most common programmable accelerators for deploying DNNs due to their high parallelization and high-speed floating point computing power. However, GPUs cannot be deployed in embedded systems as a result of their high power consumption.

Recent studies [6, 12] have been using Field Programmable Gate Arrays (FPGAs) as a more energy-efficient alternative to GPUs for deploying DNNs. FPGAs present advantages in terms of high flexibility to design dedicated hardware, fixed-point calculation, parallel computing and low power consumption. Accelerators based on Coarse Grained Reconfigurable Arrays (CGRAs) for DNNs have also been further investigated. A CGRA is a programmable hardware circuit from the same family of the FPGAs but with a lighter configuration infrastructure, resulting in less silicon area and lower cost.

The Deep Versat CGRA [7] is a configurable and customisable hardware accelerator developed for speeding-up loop-based applications. Although highly scalable, this reconfigurable processor presents limitations for the acceleration of...
CNNs. Thus, the main focus of this work is the
development of a new hardware accelerator named
VersatCNN, which is based on the former Deep
Versat but suitable for the computation of CNNs.
The improvements include the addition of a Direct
Memory Access (DMA) module for fast data trans-
fers, the deployment of vector Functional Units
(FU) for shared configurations between the same
type of FUs, the implementation of automatic ping-
pong memories and heterogeneous stages.

The second main goal is to validate and demon-
strate the VersatCNN IP core by the deployment
and acceleration of an object detector for an ambi-
tious performance of at least 30 Frames Per Second
(FPS). As a result, the source code of the origi-
nal floating-point model of the detector is firstly re-
duced for its application on an resource-constrained
embedded system and then simplified for hardware
computation, which englobes post-training fixed-
point quantization and approximation of activation
functions. The software baseline is imple-
mented on top of the IObundle System-On-Chip
(IOb-SoC) platform, which is based on a RISC-V
soft-processor.

This document is organized as follows. Section 2
introduces the background of CNNs, the YOLOv3-
Tiny detector and Deep Versat. Section 3 describes
the architecture of the IP core developed to acce-
lerate CNNs, which is inspired on the Deep Versat
CGRA. In Section 4, YOLOv3-Tiny is accelerated
using the VersatCNN IP core. Section 5 presents
the performance results of the final solution in terms
of the resource consumption, the execution time of
the detector and the comparison with other FPGA-
based works. Finally, Section 6 concludes the work
and highlights the major achievements and sugges-
tions for future work.

2. Background
2.1. Convolutional Neural Networks
CNNs are implemented as a sequence of intercon-
ected layers and consist in two stages: feature ex-
traction and classification. For feature extraction,
the network is built on repeated blocks, each com-
posed of a convolutional layer, an optional batch-
normalization layer, a non-linear layer (i.e., appli-
cation of an activation function) and an optional
pooling layer. For classification purposes, fully con-
ected layers, optionally followed by a regression
function, are typically applied after the last block
of the feature extraction stage. Modern CNN mod-
e ls add other type of layers such as shortcut, route
and upsample layers.

Convolutional layers perform 3D convolutions,
which can be seen as a set of 2D convolutions. In 2D
convolutions, a 2D kernel is overlapped and shifted
as a sliding window throughout the entire 2D input
feature map (FM), generating a 2D output FM. In
each overlap, a MAC operation is performed. In
3D convolutions, for each 3D kernel, there is a 2D
convolution between each channel of the input FM
and of the given 3D kernel. The results of the con-
volutions are summed across all the channels. The
output feature map is obtained after summing the
former result with a shared bias associated to each
3D kernel. Therefore, one output FM is created for
each 3D kernel.

The batch-normalization layer is used for speed-
up the training by normalizing the input data. Eq. 1 expresses the computation performed by this
layer for each input element, $x$, where the mean, $\mu$,
and the variance, $\sigma^2$, are statistics collected from
training and the scale factor, $\gamma$, and the shift fac-
tor, $\beta$, are parameters learned during training. $\epsilon$
is a small constant that avoids dividing by zero.

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

The pooling layer downsamples the feature maps. Each 2D channel is divided into blocks, which are
further replaced by the maximum (maxpooling) or
the mean (average pooling) value of the block. The
most common operation is a 2x2 maxpooling.

The shortcut layer skips one or more layers by
adding the output of a former layer to the input of
the current layer. The route layer concatenates
the output from a former layer with the input of
the current layer by stacking them into different
channels. The upsample layer upsamples a feature
map, typically by a factor of two.

2.2. CNN acceleration with FPGAs
The most common approaches for accelerating CNN
inference in FPGAs in previous works [6, 12] are
mainly focused on exploiting the parallelism of the
MAC operations of the convolutions and approxi-
mating the model for fixed-point computation.

The computation of each convolutional layer can
be seen as the application of four nested loops. Each
loop is associated to a source of parallelism: intra-
convolution (multiplications in 2D convolutions are
implemented concurrently), inter-convolution (mul-
tiple 2D convolutions are computed concurrently),
intra-FM (multiple pixels of a single output FM are
processed concurrently) and inter-FM parallelism
(multiple output FMs are processed concurrently).
The sources of parallelism to be exploited are de-
efined by applying loop optimization techniques such
as loop unrolling and loop tiling. Loop unrolling
consists in accelerating the execution of the loops
at the expense of resource utilization. Each loop
has an unroll factor that indicates how many times
the respective loop is parallelized. Loop tiling di-
vides the data into multiple blocks to increase the
data locality.
2.3. YOLOv3-Tiny detector

YOLOv3-Tiny [9] is the state-of-the-art object detector that presents the best trade-off between accuracy and execution time. The input image is resized at the beginning of the process flow as the detector allows different input resolutions. The YOLOv3-Tiny CNN extracts features and returns candidate bounding boxes from those features for two different scales (26x26 and 13x13). Candidate bounding boxes are then filtered based on their objectness score and the score of each class. Finally, non-maximum suppression is used to remove multiple detections of the same object and the final detections (bounding boxes and class labels) are drawn over the original input image.

The YOLOv3-Tiny CNN is composed of 13 convolutional layers, 6 maxpool layers, 2 route layers, 2 yolo layers and 1 upsample layer. All convolutional layers include batch-normalization and use Leaky ReLU (with slope of 0.1) as activation function, except from the convolutional layer exactly before of each yolo layer. The kernels are 3x3 and 1x1 to reduce the number of weights. The yolo layers apply the logistic activation (i.e., sigmoid) to some of their input channels.

2.4. Deep Versat CGRA

A CGRA is a collection of programmable FUs and embedded memories interconnected by programmable switches. The interconnections are reconfigurable at runtime to form different hardware datapaths that accelerate distinct computations for the same application.

The Deep Versat CGRA [7] is a multi-layer architecture composed of a set of Versats stacked in a ring structure. Each Versat has a data engine which consists of FUs organized in a full mesh topology and configuration module composed of (1) the Configuration Shadow Register, which stores the configuration currently being executed by the respective data engine and (2) the Configuration Register File, which holds the next configuration.

The Address Generator Unit (AGU) is the core of the data engine that controls the data access pattern within the FUs and manages the start and the end of the execution of a given run. The AGU consists of two cascaded counters capable of executing two nested loops in a single configuration. The computation of the addr output is controlled by the start, iterations, period and incr inputs according to Algorithm 1.

Deep Versat is controlled by a RISC-V soft-processor and presents some limitations for accelerating CNNs in terms of the: data transfer (driven through the soft-processor instead of a DMA); ping-pong memories (overlapping data computation and communication is not automatic); homogeneous layers; individual configurations per FU and the number of loops in the AGU (to perform a full 3D convolution in a single configuration, more loops are required).

3. VersatCNN IP Core

3.1. High-level architecture

VersatCNN is composed of two heterogeneous stages called xWeightRead and xComp, besides of an AXI-based DMA, as represented in Figure 1.

Algorithm 1: AGU output access pattern.
The function of the AXI-DMA block is to read/write data from/to the external memory. The DMA handles 256-bit wide data and allows configurable bursts for both reads and writes. It has two data native interfaces, allowing the xComp module to read and write from memory and an AXI4 interface to access the external memory. It also has a native configuration interface driven by the CPU. Like the FUs, the DMA can be configured while running, so that configurations, data transfers and FU computing can all happen simultaneously.

Each stage has a configuration module with specific configurations that are shared between the same type of FUs within the stage. These configurations are set via the RISC-V native. Apart from the internal configurations of each stage, there are global control and status registers that are common to all stages:

- **Run**: starts the execution of the configurations stored in the shadow registers of each stage.
- **Clear**: resets the configurations stored in the register files of each stage.
- **Done**: indicates the end of execution of all configurations of all the stages.

### 3.2. Detailed architecture

The detailed architecture of the VersatCNN IP core is shown in Figure 2. Unlike Deep Versat, the connections between the compute, vRead and vWrite FUs are fixed due to the regularity of the convolutional layers. The custom FUs are reconfigurable as in generic CGRAs, allowing them to form different hardware datapaths for different computations.

Each FU in the same row receives the same FM tile but a different 3D kernel, which corresponds to computing multiple output FMs in parallel (intra-FM parallelism), corresponding to the loop 3 unroll factor defined by \( n_{\text{Rows}} \). Therefore, the total number of FUs is \( n_{\text{Cols}} \times n_{\text{Rows}} \). Inside each FU, multiple 2D convolutions are computed in parallel (inter-convolution parallelism) and the loop 2 unroll factor is defined by \( n_{\text{MACs}} \). The remaining synthesis parameters give the address width of the respective module.

The dataflow is the following. The vRead FUs read data from the external memory using the DMA and store them internally. At the same time, the data in the vRead FUs, obtained from the external memory in the previous run, is read out and broadcast to columns or rows of FUs, depending on the type of vRead FU. Each custom FU computes a different 3D convolution. The computation results of the custom FUs are concatenated and stored in the vWrite FUs, while those of the previous run are written back to the external memory via the DMA.

#### 3.2.1 xWeightRead stage

Figure 3 shows the detailed architecture of the xWeightRead stage, omitting the **merge** module and the dataflow. This module is composed of a Bias vRead FU and an array of Weight vRead FUs. These units read data from the external memory using their External AGUs, write these data to their internal memories (Bias or Weights memories) and, at the same time, read previous data from their internal memories to feed the compute FUs.

The Weight vRead FU array share the Internal AGU to read the data from the internal memory. The external AGUs are on the other hand individual as each uses a different base address value. However, their configuration is shared because the base addresses are calculated in hardware. The base address of the first external AGU is configurable. The base address of the other external AGU is calculated by adding the base address of the first extern
nal AGU (WEIGHT_EXT_ADDR runtime parameter in
the figure) with the product of the external AGU
position and the address offset (WEIGHT_OFFSET run-
time parameter). This results from the kernels of
the same convolutional layer being typically stored
sequentially in the external memory and having the
same size. In spite of requiring the use of nCols-1
multipliers in the design, performing this calcu-
lation in hardware allows keeping the configuration
size independent of the number of vReads.

The weight memories are asymmetric dual-port
memories, having an external bus of 256 bits and an
internal bus of nMACs x DATAPATH_W bits as nMACs
weights are read simultaneously from the same 3D
kernel to perform inter-convolution parallelism. Re-
garding the bias vRead, as the same bias is used by
the custom FUs in the same matrix column, no in-
ternal AGU is needed, only a single read address
defined by the BIAS_START_B runtime parameter.

The runtime parameters of the xWeightRead
stage are used by the internal and external AGUs
that control the access pattern of the weights and
bias memories. For the vReads, the external AGU
controls the write address of the memories whilst
the internal AGU controls the read address. The
internal AGU is the same 2-loop AGU from Deep
Versat. The external AGU, represented in Figure 4,
is a new module that handles data exchanged be-
tween the external and internal memories.

![Figure 4: Interface signals of the external AGU.](image)

In this module, the communication with the ex-
ternal memory is done through the native external
memory interface where the address is calculated
by adding a base value (ext_addr parameter) with
an offset value. The offset is calculated by using
another 2-loop AGU inside the external AGU. In
turn, the communication with the internal mem-
ory is done via the native internal memory interface
where the address is determined by adding a base
value set by the int_addr parameter with an off-
set calculated by using a sequential counter inside
the external AGU. The direction parameter indi-
cates the direction of the data flow. For the vReads,
the direction parameter is hard-wired to zero which
means that the data is read from the external mem-
ory and written into the internal memory.

### 3.2.2 xComp stage

The xComp stage is composed of an array of FM
Tile vRead FUs, a matrix of custom FUs and an ar-ray of vWrite FUs. The vRead units operate analog-
ously to the vRead FUs from the xWeightRead
stage. Each custom FU receives a bias, weights and
pixels from the FM tile to compute mainly 3D con-
volutions. The vWrite units write the results to
their internal memories and read the previous re-
sults to be sent back to the external memory.

### 3.2.2.1 FM Tile vReads

The vReads for the input FM tiles present a scheme
similar to the vReads for the weights, namely with:
the calculation of the base address of each external
AGU with nRows-1 multipliers; the shared config-
urations between the external AGUs; the use of a
single internal AGU by the asymmetric dual-port
memories that store the FM tiles and the auto-
matic ping-pong operation of those memories. Fig-
ure 5 shows the detailed architecture of the xComp
vReads, omitting the calculation of the base ad-
resses of the external AGU, the merge module and
the dataflow for simplification purposes.

![Figure 5: xComp vReads detailed architecture.](image)

The read address of the memories can be con-
figured to come from the internal AGU or from
the values stored in other memory. The selection
is made by the TILE_EXT runtime parameter via a
multiplexer. This parameter is useful when the data
access pattern is not regular and cannot be deter-
mined by the AGU. The pattern memory is also

---

5
linked to an external AGU for pre-loading data access patterns from the external memory and to the same internal AGU as the FM tile memories when being read to address them. To perform a 3D convolution in a single run, the access pattern of the FM tile memories requires more than 2 loops. Hence, the internal AGU of the xComp vReads was improved to support 6 loops by cascading 3 AGUs, which adds 2 more sets of the incr, iterations, period and shift configuration parameters.

3.2.2.2 Custom FU

The detailed architecture of the custom FU is represented in Figure 6. The reconfigurable interconnections inside this module allow to form different datapaths to accelerate different CNN layers (e.g., convolutional with or without bias, maxpool) and activation functions (e.g., Leaky ReLU, sigmoid) individually or even in the same run.

The IP core implements the activation functions after the convolution. The leaky runtime parameter enables the leaky activation, which is implemented with 2 adders, 1 multiplexer and shifters. The sigmoid runtime parameter enables the sigmoid activation, which is implemented by means of simple comparators, multiplexers, adder/subtracters and a priority encoder. In case the sigmoid activation is not applied to all output channels, the mask runtime parameter is a second enable for the sigmoid computation but is individual for each custom FU in the matrix row. After the optional activation blocks, the result is shifted considering the value of the shift runtime parameter and the quantization format of the results.

The IP core was designed to allow the computation of the convolutional and maxpool layers in the same run. The computation of the maxpool, which is enabled by the maxpool runtime parameter, is performed with: a 2-bit counter to handle 2x2 blocks of pixels; a comparator to find the maximum value in the 2x2 block and a multiplexer to select that value. Note that the enable of the counter is also controlled by the AGU. The maxpool can also be performed standalone (without performing convolutions in the same run) by bypassing the pixels from the FM tile to the input of the maxpool computation. This configuration is enabled by the bypass runtime parameter. The last runtime parameter regards to the option of bypassing the result of one of the MACs to the output of the custom FU by enabling the bypass_adder runtime parameter. It is used when only needing to compute individual accumulations with a single MAC.

3.2.3 vWrite FUs

The runtime parameters for the xComp vWrites are used by the internal and external AGUs that control the access pattern of the memories that store the computation results. For the vWrites, the internal AGU controls the write address of the memories whilst the external AGUs control the read address of the memories. Therefore, the direction parameter of the external AGUs is hard-wired to one, which indicates that the data is read from the internal memories and written into the external memory. As the vReads, the vWrites: require nRows−1 multipliers for the calculation of the base address of each external AGU; use a single internal AGU shared by all memories and share the runtime parameters between the external AGUs.

3.2.3 AXI-DMA

The AXI-DMA module consists of two finite state machines (one for the reads and another for the writes) that convert the requests of the vReads and
vWrites (native interface) to AXI4 read and write transactions (AXI4 interface).

For the read transactions (vReads), the runtime parameters account for the total number of 256-bit aligned transactions in a single run. The AXI4 protocol supports a maximum of 256 transfers per burst. Therefore, the DMA contains an internal counter, initialized at the beginning of the configuration run with the total number of required transactions, that decrements each time a transaction is done in order to determine the number of transactions in each burst. For instance, if the total number of 256-bit aligned transactions is 500, the first burst will have 256 transactions whilst the second burst will have 244 transactions (500–256).

The runtime parameter for the write transactions (vWrites) accounts for the total number of bytes (not 256-bit transactions) to transfer in a single run. The difference regards to the fact that the DMA write may require unaligned transactions, i.e., being able to write from any memory address any number of bytes. Hence, the DMA also includes an aligner module that manages the data bytes and the strobe to align the data with the DMA 256-bit databus.

3.3. Operation

The VersatCNN IP core is first configured by writing to the configurable register files and then run by writing a command to the run control register. The run command executes the configurations transferred from the register files to the shadow registers. As a result, the next run can be configured during the current run without affecting its operation. Excepting the first and last two runs, the IP core can read and write to the external and internal memories, compute and be configured for the next run, all in parallel.

The operations are implemented in a pipelined fashion which means that, after the first two runs, different data is being read, computed and written in the same run. The management of the configurations in pipeline fashion is challenging in software as the programmer would need to program all the configurations at the same time taking into account the priority defined by the merge module, resulting in several lines being repeatedly read from the external memory in the same run.

To optimize the read process, each tile vRead is coupled with a comparator that compares the address of the vRead with the address at the databus interface (which corresponds to the address of the vRead that earned the priority in the merge module) and, in case they are equal, a multiplexer chooses the data coming from the databus. Consequently, the common lines between vReads are stored at the same time, saving communication time in the run.

4. Implementation of YOLOv3-Tiny

4.1. Optimizations for hardware implementation

To be able to run the software application on an resource-constrained embedded system, additional optimizations, such as linear approximation of activation functions, batch-normalization folding and post-training quantization, were deployed.

The slope value (0.1) of Leaky ReLU is approximated by replacing the multiplication by a sum of multiple right shifts of the input value, as shown in Eq. 2. The sigmoid was implemented by the piecewise linear approximation in [13].

\[
x \times 0.1 \approx (x >> 4) + (x >> 5) + (x >> 7)
\]  

(2)

The batch-normalization folding consists of a linear transformation to fold the parameters of the batch-normalization layer into the preceding convolutional layer. The pre-trained floating-point weights \(w\) and biases \(b\) are updated to their new values \(w'\) and \(b'\) according to Eq. 3.

\[
w' = \frac{\gamma \times w}{\sqrt{\sigma^2 + \epsilon}} \quad b' = b - \frac{\mu \times \gamma}{\sqrt{\sigma^2 + \epsilon}}
\]  

(3)

The CNN computation is typically approximated to fixed-point format for inference in FPGAs. The fixed-point format for the weights, biases and FMs in each layer is chosen by selecting the minimum number of bits needed for the integer part to avoid overflow, leaving the remaining bits for the fractional part. All values were quantized using 16 bits. The final fixed-point model resents a \(mAP_{50}\) drop of 2.1 in comparison with the original floating-point model for the MS COCO 2017 test dataset.
4.2. IOb-SoC-Yolo

IOb-SoC [3] is an open-source RISC-V-based System-On-Chip platform developed by IObundle. The system is composed of a low-performance RISC-V soft-processor to control the slaves (i.e., memory sub-system and peripherals). The slaves include: boot controller (runs bootloader), internal memory (stores firmware), external memory, timer (measures the time performance of the application), UART (for the bootloader and debugging) and Ethernet (transfer big data files). The VersatCNN is integrated as another peripheral in the SoC platform (to accelerate YOLOv3-Tiny) which is renamed to IOb-SoC-Yolo.

4.3. Performance of the software baseline

The software baseline is divided in 4 sections: setup (peripherals initialization and preparation of the data in the external memory), pre-CNN, CNN and post-CNN. The execution time of the software-only version running on the IOb-SoC platform (using O3 optimizations) at 143MHz is 969 seconds (above 16 minutes). The target frame rate for this work is 30 FPS, which corresponds to a total execution time of 33.3 ms. Therefore, all CNN layers need to be accelerated in hardware. The pre-CNN process takes approximately 1 second on the CPU and must also be accelerated in the same hardware as the CNN. The post-CNN process is fast enough in software, except the draw detections method, which can be accelerated in hardware using a DMA engine.

4.4. Accelerating YOLOv3-Tiny with VersatCNN

Most of the synthesis parameters that determine the internal architecture of the VersatCNN IP core are defined by the loop unroll and loop tiling factors chosen to accelerate the YOLOv3-Tiny network. Previous works performed design space exploration in order to choose the factors that achieve the maximum computational throughput. This work follows a slightly different approach by theoretically choosing the factors that allow to achieve a target frame rate of 30 FPS (i.e., 33.3 ms) taking into account the characteristics of the YOLOv3-Tiny CNN.

The execution time for the computation of the convolutional layers depends on the parallelism factor and the clock frequency. The parallelism factor must be carefully chosen considering the CNN characteristics for not leading to the underutilization of the MAC resources: Inter-FM parallelism factor defined by $nCols$ is 16 as all the CNN layers have a number of kernels multiple of 16; Intra-FM parallelism factor defined by $nRows$ is 13 as all layers present an input FM with a height multiple of 13; Inter-convolution parallelism factor defined by $nMACs$ is 4 as all layers present a number of input channels multiple of 4. The total parallelism factor chosen for the IP core is then $832$ ($16 \times 13 \times 4$), which leads to an estimated execution time of 23.4 ms for a clock of 147 MHz. In turn, the tiling factor of each layer is chosen so that the communication time is below the computation time in each run.

5. Results

The development board available for this work is the Kintex UltraScale KU040 [2], which includes a Xilinx XC7KU040 FPGA. The vWrite and the bias memories are implemented using LUTRAMs whilst the other vRead memories are implemented with BRAMs. Each MAC of the VersatCNN IP core is implemented in the FPGA by one DSP with 4 pipeline stages, no pre-adder and the ALU configured as an accumulator.

5.1. Resource consumption

Table 1 presents the resource consumption in terms of the FPGA primitives of the IOb-SoC-Yolo system, where most of the resources are occupied by the hardware accelerator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>VersatCNN IP</th>
<th>IOb-SoC-Yolo</th>
</tr>
</thead>
<tbody>
<tr>
<td>36Kb BRAM</td>
<td>339</td>
<td>382.5 (64%)</td>
</tr>
<tr>
<td>FF</td>
<td>86,319</td>
<td>110,988 (23%)</td>
</tr>
<tr>
<td>LUT Logic</td>
<td>104,655</td>
<td>119,166 (49%)</td>
</tr>
<tr>
<td>LUT memory</td>
<td>16,792</td>
<td>19,780 (18%)</td>
</tr>
<tr>
<td>DSP</td>
<td>871</td>
<td>878 (46%)</td>
</tr>
</tbody>
</table>

Overall, the design requires less than half of the resources available at the target device, with the exception of the BRAMs, which are used as on-chip memory to store all channels from 3D kernels and input FM tiles in order to perform 3D convolutions in a single run. The implementation of shared configurations between the same type of FUs allowed the system to be scalable in terms of both Flip-Flop and LUT consumption. Only 46% of the DSPs are used, thus, the parallelism factor could still be doubled from 832 to 1664 if requiring a higher target frame rate.

5.2. Execution time

The total execution time of the YOLOv3-Tiny detector implemented over the IOb-SoC-Yolo platform is 30.9 ms, exceeding the target frame rate of 30 FPS. In comparison with the software baseline, the pre-CNN was accelerated from 1s to only 3ms and the drawing detections method from the post-CNN was improved from approximately 12ms to nearly 1.4ms, both mainly due to the reduction of the communication time between the FPGA and the external memory by using the DMA engine inside the IP core. The highest speed-up was achieved for the acceleration of the CNN from 968 seconds to only 24.4ms.

Table 2 compares the execution time of the fixed-point model of the YOLOv3-Tiny detector implemented over the IOb-SoC-Yolo platform with the
original floating-point model from Darknet executed in both CPU and GPU. IOb-SoC-Yolo is nearly 27 times faster than the CPU version and only 2 times slower than the GPU version, being however more suitable for embedded systems.

Table 2: YOLOv3-Tiny performance per platform.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time (ms)</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (Intel i7-8700)</td>
<td>828.3</td>
<td>1.2</td>
</tr>
<tr>
<td>GPU (RTX 2080 Ti)</td>
<td>15.4</td>
<td>64.9</td>
</tr>
<tr>
<td>FPGA (IOb-SoC-Yolo)</td>
<td>30.9</td>
<td>32.4</td>
</tr>
</tbody>
</table>

5.3. Comparison with FPGA implementations

At the time of writing of this document, three others implementations of the YOLOv3-Tiny detector in FPGA are reported in the literature. All three implementations consist of hardware/software co-designs. The IOb-SoC-Yolo is compared with these implementations in Table 3.

In [8], the YOLOv3-tiny model is first trained using the Caffe framework over a dataset for pedestrian signalling and then quantized with 8-bit fixed-point. The backbone network is accelerated by the FPGA whilst the detection layers are handled in software by the hard processor. The author claims a throughput of 104.2 FPS without detailing the hardware architecture or resource consumption.

[1] applies batch-normalization folding and post-training quantization of 18 bits. Only the convolutions are handled in hardware and all the other layers and activation functions are implemented in software. The hardware architecture exploits the inter-FM, inter-convolution and intra-convolution parallelisms with a total parallelism factor of 2304. In comparison with IOb-SoC-Yolo, the total parallelism factor is over 2.5x higher, which would justify a higher throughput. However, the only performance metric reported is the number of MAC operations per second, calculated from the product between the number of DSPs and the frequency. Therefore, the actual throughput is not reported and no fair performance comparison can be done between the two works.

[15] accelerates all YOLOv3-Tiny layers in hardware. In comparison with IOb-SoC-Yolo, the intra-convolution parallelism (loop 1) is exploited instead of the intra-FM (loop 3), the data is also quantized with 16 bits and the throughput is about 17x lower. Note that the Zynq 7020 device has between 4 and 8x less hardware resources than the UltraScale XCKU040 (depending on the specific FPGA primitive). Both works perform 16-bit MAC operations and, based on the MAC operations per second, IOb-SoC-Yolo has better performance and also better area efficiency in terms of LUT and DSP consumption.

All the other works only focus on the acceleration of the CNN part of the YOLOv3-Tiny detector. In turn, this work executes the full process flow of the detector by adding the image resize prior to the CNN and the drawing of the detections after the CNN. The reconfigurable interconnections of the VersatCNN IP core allow to form different datapaths to accelerate more than only CNNs. As a result, the core is also able of accelerating the pre and post-processing parts of the detector.

6. Conclusions

This work presents the development of a new extension for the Versat reconfigurable processor called VersatCNN, which is optimized for the acceleration of CNNs. This hardware accelerator is added as another peripheral in the IOb-SoC platform with the purpose of accelerating the YOLOv3-Tiny object detector. The IP core is parameterized taking into account the characteristics of the CNN and the available resources in the target device, achieving a performance over 30 FPS.

The source code from the Darknet framework [10] is reduced and adapted for resource-constrained embedded systems, which includes hardware optimizations such as the approximation of the activation functions, batch-normalization folding and post-training dynamic quantization. The fixed-point model presents a mAP drop of only 2.1 in comparison with the original floating-point model. The software baseline is executed on the IOb-SoC platform and presents an execution time over 16 minutes and the profiling results show that the pre-CNN all CNN layers and the drawing detection method of the post-CNN must be accelerated in hardware to achieve the target frame rate.

The Deep Versat CGRA presents limitations for the acceleration of CNNs, which lead to development of VersatCNN for efficient CNN computation. The improvements involve the implementation of vector FUs that share configurations between the same type of FUs, the integration of a DMA for fast data transfers, heterogeneous stages, automatic ping-pong memories and higher loop-level AGUs. The custom MAC-based FUs are structured in a matrix form to exploit three types of parallelism (Inter-FM, Intra-FM and Inter-Convolution) and to enhance pixel and weight sharing.

The IP core is tested for the acceleration of the YOLOv3-Tiny detector in a FPGA. Its architecture is pre-configured with a total of 16 weight memories of 32kB, 13 FM tile memories of 64kB and 208 custom FUs, each with 4 MACs, for a total parallelism factor of 832. The yolo, upsample and most of the maxpool layers are executed alongside the previous convolutional layer. The pre-CNN and the drawing of the detections from the post-CNN are also accelerated by the IP core. As a result, the system composed of IOb-SoC and VersatCNN achieves a
Table 3: Comparison of FPGA-based implementations of the YOLOv3-Tiny detector.

<table>
<thead>
<tr>
<th></th>
<th>UltraScale+ XCZU9EG</th>
<th>Virtex-7 XC7VX485T</th>
<th>Zynq 7020</th>
<th>UltraScale XCZU040</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>-</td>
<td>200</td>
<td>100</td>
<td>143</td>
</tr>
<tr>
<td>LUT (K)</td>
<td>-</td>
<td>49</td>
<td>26</td>
<td>119</td>
</tr>
<tr>
<td>BRAM</td>
<td>-</td>
<td>70</td>
<td>93</td>
<td>384</td>
</tr>
<tr>
<td>DSP</td>
<td>-</td>
<td>2304</td>
<td>160</td>
<td>878</td>
</tr>
<tr>
<td>Images/s</td>
<td>104.2</td>
<td>-</td>
<td>1.9</td>
<td>32.4</td>
</tr>
<tr>
<td>Unrolled loops</td>
<td>1.2,4</td>
<td>-</td>
<td>1.2,4</td>
<td>2.3,4</td>
</tr>
<tr>
<td>Precision (bits)</td>
<td>8</td>
<td>18</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>GMAC/s</td>
<td>-</td>
<td>-</td>
<td>5.3</td>
<td>90</td>
</tr>
<tr>
<td>MMAC/s/kLUT</td>
<td>-</td>
<td>-</td>
<td>203.8</td>
<td>756.3</td>
</tr>
<tr>
<td>MMAC/s/DSP</td>
<td>-</td>
<td>33.1</td>
<td>102.5</td>
<td></td>
</tr>
</tbody>
</table>

performance of 32.4 FPS for the full YOLOv3-Tiny detector, which shows that VersatCNN is a valid solution for accelerating CNN-based networks.

Future developments on this work may include the deployment of a generic infrastructure to demonstrate the object detector in real-time, the improvement of the performance of the soft-processor and the Ethernet module and further validation of the VersatCNN IP core by accelerating other CNN-based networks.

References


