A Parameterisable FPGA-tailored Architecture for YOLOv3-tiny

Zhewen Yu and Christos-Savvas Bouganis

Department of Electrical and Electronic Engineering, Imperial College London
{zhewen.yu18,christos-savvas.bouganis}@imperial.ac.uk

Abstract. Object detection is the task of detecting the position of objects in an image or video as well as their corresponding class. The current state of the art approach that achieves the highest performance (i.e. fps) without significant penalty in accuracy of detection is the YOLO framework, and more specifically its latest version YOLOv3. When embedded systems are targeted for deployment, YOLOv3-tiny, a lightweight version of YOLOv3, is usually adopted. The presented work is the first to implement a parameterised FPGA-tailored architecture specifically for YOLOv3-tiny. The architecture is optimised for latency-sensitive applications, and is able to be deployed in low-end devices with stringent resource constraints. Experiments demonstrate that when a low-end FPGA device is targeted, the proposed architecture achieves a 290x improvement in latency, compared to the hard core processor of the device, achieving at the same time a reduction in mAP of 2.5pp (30.9% vs 33.4%) compared to the original model. The presented work opens the way for low-latency object detection on low-end FPGA devices.

Keywords: YOLOv3-tiny · FPGA · object detection

1 Introduction

The object detection technology deals with the problem of detecting instances of objects in images and videos. Applications of this technology can be found in the deployment of advanced intelligent systems like Advanced Driver Assistance Systems (ADAS) and video surveillance. Accurate object classification and identification of the objects' position are often required, as this information forms the basis for further processing and decision making in the rest of the application's pipeline.

Recently, capitalising on the recent advances in machine learning, and more specifically on the development of deep neural networks, researchers and practitioners have developed powerful object detection systems that can provide accurate detection in a number of challenging situations. Furthermore, in cases where low latency of processing is required, work in the area has moved away from scanning the image in multiple positions and applying image classifiers (i.e. casting the problem of object detection to the classification problem over multiple windows) to combining the above distinct steps in a single pipeline, usually based on a deep neural network.

Early works towards this direction include R-CNN [4], Fast R-CNN [3] and Faster R-CNN [15]. These works implement the object detection by two distinct parts, region proposal selecting possible candidates that include an object and a deep neural network responsible for the classification of these regions. As resource sharing between the two parts is limited, the above approach exhibits usually high computational loads and detection latency.

In an attempt to provide object detectors with lower computational requirements and enable object detection in low-power devices, research has focused on the one-step approach, where the bounding boxes around the objects are predicted directly though the DNN rather than having a separate region proposal step[19]. Two of the most popular such frameworks are the Single Shot MultiBox Detector (SSD)[7] and the YOLO (You only look once)[13].

SSD is based on a VGG16 network and has been extended by custom convolution layers in order to generate bounding boxes. SSD uses a set of predefined anchor boxes for detection at various scales impacting the framework's precision and computational load.

The YOLO framework relies on a single DNN, DarkNet, in order to predict both the position of the objects (i.e. bounding boxes) as well as their classification. Early versions of the YOLO approach exhibited low computational loads by trading the classification precision for low latency, which led to their deployment in embedded systems. The most recent version of YOLO is YOLOv3, which adopts a deeper neural network than its predecessors, achieving more accurate classifications. Currently, YOLOv3 demonstrates its advantages of both low latency and high classification precision over other competitors [14].

In cases where the deployment of an object detection system is required in an embedded form for real-time applications, such as an ADAS system, low power and latency considerations have pushed the designers to solutions that are based on low-power FPGA and mobile GPU platforms. Because generic mobile processors usually do not provide the necessary performance under the desired power envelopes.

This work addresses the challenging problem of deploying YOLO in a low-power FPGA device, and more specifically it targets the mapping of YOLOv3-tiny, a variant of YOLOv3 version for embedded systems. The paper's novel contribution is a latency-optimised parameterisable architecture tailored to YOLOv3-tiny workload that can be tuned to the resource availability of any targeted FPGA device. To enable the above, a parameterisable architecture is developed and implemented using Vivado HLS. Performance and resources models are derived that guide the Design Space Exploration (DSE) phase for identifying design points that optimise the latency of the system, meeting at the same time the resource constraints.

To the best of the authors' knowledge, this is the first work that addresses this problem when a low-power and limited resources FPGA device is targeted and use of off-chip memory is required to store the parameters and intermediate results of the network, enabling the deployment of YOLOv3-tiny in scenarios with extremely limited resources.

The rest of the paper is organised as follows. Section 2 describes network architecture of YOLOv3-tiny and challenges in its mapping to an embedded system. Section 3 elaborates the proposed architecture and explains how YOLOv3-tiny is mapped onto the accelerator. Section 4 focuses on the resource and performance models which enable the Design Space Exploration described in Section 5. Section 6 introduces the wordlength optimisation used in the architecture, where Section 7 provides a discussion on the evaluation of the system. Finally, Section 8 concludes the paper.

2 Background

2.1 YOLOv3-tiny Network

YOLOv3-tiny is a light-weight version of YOLOv3. It exhibits a reduced number of layers compared to YOLOv3, allowing its deployment to resource-constraint devices. The reduced number of layers leads to lower computational load and inference latency with a penalty on the object detection precision.

YOLOv3-tiny accepts an RGB image of 416×416 resolution as input. Contrary to YOLOv3, YOLOv3-tiny predicts bounding boxes at two different scales only. The first scale divides the input image into 13×13 grids, while the second operates on 26×26 grids. The framework generates three bounding boxes in every grid. The network outputs a 3d tensor containing information on the bounding box, the objectness confidence and the class predictions.

The network mainly utilises five types of layers; Convolutional, Max pooling, Route, Upsample and Yolo layer. Route layers are responsible for creating different flows in the network, where Upsampling is used to support multiple detection scales. The Yolo layer is responsible to generate the output vector. Fig. 1 shows the dataflow of the network alongside with the number of operations required for convolution.

2.2 Mapping YOLO-based networks to FPGAs

Current toolflows for automated mapping of CNNs to FPGAs do not support the specialised computational layers of YOLO [16]. As such, a number of works have focused on customised designs for deploying various versions of the YOLO network onto FPGAs. Some perform a faithful mapping of YOLO while others introduce certain approximations to tailor the hardware. Wei et. al[18] presented an FPGA-based architecture for the acceleration of YOLOv2-tiny, where the parameters of the layers are required to be stored on local memory in advance. In their architecture, the Leaky ReLU activation is replaced with ReLU for reducing the resource consumption. The authors reported a latency per inference of 52 ms on Zynq 7035.

The acceleration of YOLOv2-tiny is also targeted in Liu et. al[6]. Their system combines adjacent layers in order to reduce the communication cost between the host and the device. In their approach, a fixed point number representation

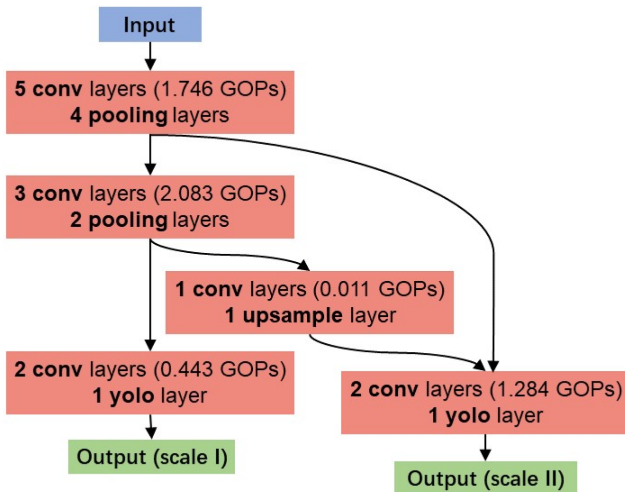
Input
5 conv layers (1.746 GOPs)
4 pooling layers
3 conv layers (2.083 GOPs)
2 pooling layers
1 conv layers (0.011 GOPs)
1 upsample layer
2 conv layers (0.443 GOPs)
1 yolo layer
2 conv layers (1.284 GOPs)
1 yolo layer
Output (scale I)
Output (scale II)

Fig. 1. Dataflow of YOLOv3-tiny

is utilised, where the inputs and outputs are kept at 16 bits while intermediate multiplication products are stored at 32 bits. Their system achieves 69.2 fps using Arria 10 GX1150.

Wai et. al[17] also targeted the acceleration of YOLOv2-tiny. In the proposed solution, the batch normalisation and convolution layers are merged together. In their system, the computation of the convolutions is based on a General Matrix-Matrix Multiplication (GeMM) core that has been designed via OpenCL. The proposed design can be parameterised by tuning the multiplication block size. Their work achieves 3.06 fps using Cyclone V PCIe device.

Nakahara et. al[9] proposed a modified version of YOLOv2 which was termed Lightweight YOLOv2. In the proposed network, the convolution layers for feature extraction are replaced with binary multiply-add operations. Furthermore, a Support Vector Machine algorithm (SVM) is responsible for the prediction of the objects' coordinates and classifications. Using a ZCU102 device, their system achieves 40 fps, with a modest drop in accuracy (1.5pp).

Nguyen et. al[10] proposed Sim-YOLOv2, a quantised version of YOLOv2, and its implementation on FPGA. The proposed architecture focuses on maximising the throughput of the system, and as such each layer of the network is mapped to a dedicated hardware block. Weights are stored on chip to minimise off-chip data transfers. Due to the requirement of on-chip storage, a Virtex-7 VC707 device is targeted. The latency reaches 9.15 ms at the cost of 18.29 W power consumption.

2.3 Challenges and Target

Mapping YOLOv3 or YOLOv3-tiny to an FPGA device brings new challenges. In YOLO and YOLOv2, the networks are trained for VOC [1, 2] dataset and are able to address twenty different object classes. YOLOv3 is trained for COCO2014[5] which instead targets 80 classes, and as such it exhibits higher computational load.

The target of this work is the design of a latency optimised and FPGA-tailored architecture that can be customised to the available FPGA resources in order to accelerate the inference stage of YOLOv3-tiny model. The previously mentioned works are addressing the problem of mapping YOLOv2, and its variants to an FPGA device. To the best of the authors' knowledge, the only work that addresses the mapping of the newest model, the YOLOv3-tiny, is LogicTronix [8], where a Xilinx DPU-DNNDK was used for its acceleration using a ultra96 FPGA device, achieving 30 fps. Even though the work quantises the model, no evaluation or any accuracy result is reported. The proposed framework targets lower-end devices that have considerably fewer resources.

3 Proposed Architecture

The proposed architecture is tailored for the execution of YOLOv3-tiny model, providing as such hardware support for the newly introduced special Yolo layer. The developed accelerator is utilised for the execution of all layers of the network through a run-time parameter setting. The proposed architecture is tailored and compile-time parameterisable in HLS to target low-end FPGA devices with limited resources, and as such no hard constraints on adequate on-chip memory for storing of the data are imposed by the system.

3.1 System Overview

Fig. 2 provides a high-level picture of the proposed architecture. The FPGA Hardware Accelerator denotes the proposed FPGA architecture and consists of a three-stage pipeline, where each stage corresponds to a layer of YOLOv3-tiny network. The Accelerator is controlled by the ARM processor who is responsible for the overall control of the system. The data and weights are transferred between the Accelerator and the off-chip memory through DMA interfaces.

The FPGA accelerator consists of a three-stage pipeline. The first stage of the pipeline supports the execution of the convolution layer, whose output is accumulated in the second stage of the pipeline. Depending on the network structure that is executed at a given time, the accumulation results are sent for further processing in the Max pooling, Upsample or Yolo layer.

3.2 Module Design

The FPGA Hardware Accelerator consists of five main computational blocks; the convolution, accumulation, max pooling, upsample and yolo blocks.

Convolution block: The block performs direct convolutions of N_{in} input channels with the corresponding kernels and produces N_{out} output channels.

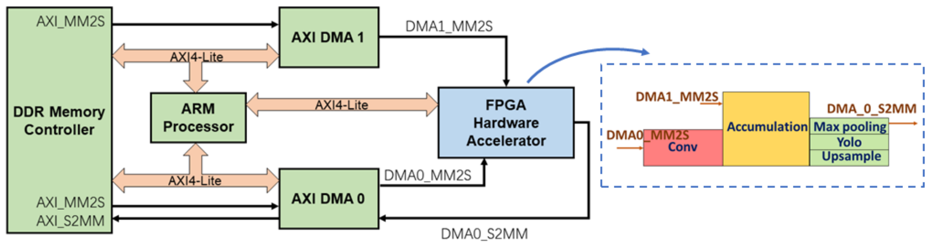

Fig. 2. System-level architecture of this work

Internally the block contains three main sub-modules, including an input line buffer, convolution kernels and an output buffer.

The compile-time (synthesis-time) known parameters of the block are

$$\{N^{max}, f_w^{max}, K_c^{max}, p_c\}$$

which define the maximum number of input and output channels, the maximum width of the input feature map, the maximum kernel size, and the number of parallel compute units in the block respectively.

The adopted architecture of the block provides flexibility on trading off resources for performance. p_c provides the option to control the unrolling factor of computing the convolutions, as well as the way that the input and output channels are interleaved. As Fig. 3 shows, p_c can also be expressed as the product of parallelism among output channels ($p_{c,1}$), parallelism among input channels ($p_{c,2}$) and parallelism inside convolution kernels ($p_{c,3}$). The maximum number of input and output channels (N^{max}) depends on the available FPGA resource and DMA bandwidth of the platform. Both p_c and N^{max} are tuned to the targeted device during the Design Space Exploration stage.

During run-time, the operation of the block is tuned by the ARM processor through the following parameters

$$\{N_{in}, N_{out}, f_h, f_w, K, S\}$$

where f_h and f_w are the padded height and width of input feature maps, K is the kernel size, and S is the kernel stride.

Accumulation block: The block is designed to allow input channels folding (Section 3.4), supporting the convolution block. The module accumulates outputs of multiple sub-layers, and is parameterised during the compile time through the parameters

$$\{N^{max}, f_w^{max}, p_a\}$$

which define the maximum number of input channels, the maximum width of the input feature map, and the number of parallel compute units in the block respectively. During run-time, the operation of the block can be customised through the following parameters:

$$\{N_{in}, f_h, f_w, E_{bias}, E_{leaky}\}$$

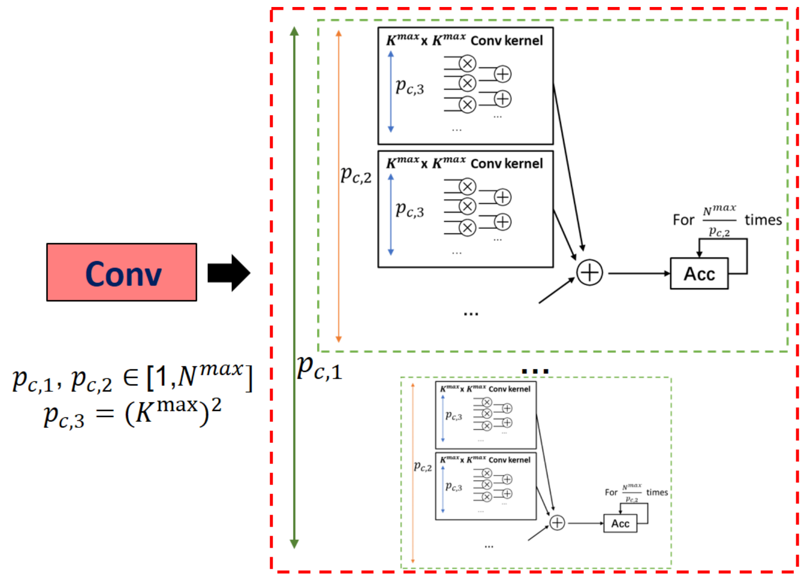
Conv
K^{max} x K^{max} Conv kernel
$p_{c,3}$
$p_{c,2}$
$p_{c,1}$
For $\frac{N^{max}}{p_{c,2}}$ times
Acc
$p_{c,1}, p_{c,2} \in [1, N^{max}]$
$p_{c,3} = (K^{\max})^2$

Fig. 3. Parameterisation of the Convolution block

E_{bias} controls whether the bias is added to the accumulated results. E_{leaky} is a parameter that enables Leaky ReLU activation function.

Yolo block: The block implements the functionality of the YOLO layer which is mainly composed of sigmoid activation. The sigmoid function is implemented as a fixed point division operation and an exponential module. The compile-time parameters of the block are

$$\{N^{max}, f_w^{max}, p_y\}$$

p_y defines the number of parallel compute units in the block. During run-time, the block is parameterised by setting the following parameters:

$$\{N_{in}, f_h, f_w\}$$

In the proposed implementation, the maximum and average error of the sigmoid implementation are 0.39% and 0.19% respectively.

Max pooling block and **Upsample block:** These two blocks are responsible for the downsampling and upsampling operations respectively. Two blocks can be parameterised at compile and run-time similar to the Yolo block.

3.3 Network Mapping to FPGA Hardware Accelerator

Given a configuration of the proposed FPGA architecture, the workload of YOLOv3-tiny is mapped onto the accelerator. As computational hardware blocks are time-multiplexed for the successful execution of the workload, certain transformations of the network need to take place in order to schedule the execution of the tasks on the accelerator.

Channels Folding: At synthesis time, the maximum number of channels N_{max} that can be processed by the proposed system is determined. When N^{max} is smaller than N_{in} or N_{out}, the computation of a layer has to be spilt into multiple sub-layers. The computation of a convolution layer having N_{in} input channels and N_{out} output channels can be expressed as:

$$g_j = \sum_{i=1}^{N_{in}} f_i * w_{i,j} + b_j, \quad with \quad j \in [1, N_{out}] \tag{1}$$

f_i is the ith input channel, and g_j is the jth output channel. $w_{i,j}$ and b_j are weights and biases of the convolution layer. When input channels are folded by a factor of $F_{in} = \lceil \frac{N_{in}}{N^{max}} \rceil$, (1) is turned into

$$g_j = \sum_{i=1}^{N^{max}} f_i * w_{i,j} + \sum_{i=N^{max}+1}^{2N^{max}} f_i * w_{i,j} + ... + \sum_{i=(F_{in}-1)N^{max}+1}^{N_{in}} f_i * w_{i,j} + b_j \tag{2}$$

By folding the output channels, g_j is divided into F_{out} sub-layers, with $F_{out} = \lceil \frac{N_{out}}{N^{max}} \rceil$, each containing up to N^{max} output channels.

Kernel Size Padding: The size of the kernels varies across the layers of the network. In this work, the maximum required kernel size (K^{max}) is actually implemented. The computation with smaller kernel sizes is performed by embedding the kernel in the maximum supported kernel. Necessary padding is applied during the process.

Layer Fusion Computation: As the work targets devices with limited resources, the mapping of the network onto the proposed architecture is broken down into smaller tasks that are supported by the proposed architecture. As such, a policy is needed to guide the segmentation of the computational load in smaller chunks such as the system's performance is optimised. The policy that has been adopted in this work is to use the convolution layers as points of boundaries for the division of the network workload into smaller chunks. As such, network layers between two boundary points are bounded together as a batch, which is referred as "**layer batch**" for the rest of this paper.

As such, the original network structure $\mathbf{N}$ is transformed into a series of layer batches $\mathbf{N}_i$, where for each layer batch i the run-time parameters of the architecture are defined as follows:

$$\mathbf{N}_i = \{F_{in}, F_{out}, N_{in}, N_{out}, f_h, f_w, K, S, \mathbf{E}\}$$

, where $\mathbf{E}$ provides information regarding the activation or not of a specific block inside the accelerator in the current layer batch. During the inference stage, the smallest scheduled task is a layer batch, where the data are streamed in and out of the accelerator. Using the above approach, off-chip memory accesses are omitted between layers inside the same batch.

3.4 System Processing Flow

During the inference stage, the ARM processor acts as a master and controls the inference process. The computation of the network is broken down to smaller components, the layer batches, which are scheduled by the processor and executed sequentially. Fig. 4 captures the processing flow for a single layer batch.

More specifically, the ARM processor firstly sets the parameters for each individual block in the hardware accelerator and configures the DMA modules. Then, it initiates the weight loading and input data transfers via DMA streaming onto the Hardware Accelerator. The FPGA Acceleration block starts processing the data and the output data are transferred back to off-chip DDR memory. The necessary invalidation of the corresponding cache region is performed by the processor, ensuring correct data transfer.

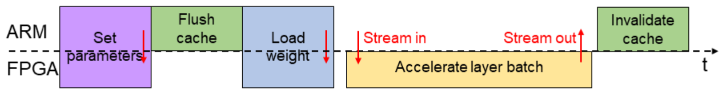

Fig. 4. System processing flow for a layer batch

4 Latency and resource estimations

Analytical models have been derived to provide resource and latency estimates of the system under a specific configuration and load. Utilising the derived models, a design space exploration phase is possible for the identification of Pareto-optimal design point in the latency-resource space.

4.1 Hardware Latency Model

As the proposed architecture of the FPGA accelerator has a pipeline structure, the initiation interval II_{sys}, i.e. the number of clock cycles before a new input can be processed, is dictated by the slowest active block. The initiation interval for each individual block is denoted by II_b, where b={*convolution, accumulation, max pooling, upsample, yolo*} and it is a function of:

$$II_b = \max\left(\frac{p^{max}}{p_b}, r_b^{OtoI}, OP_b\right) \tag{3}$$

where p_b^{max} is the maximum number of parallel compute units in theory and p_b is the actual number of units that finally implemented, r_b^{OtoI} is the ratio of output to input data transfer size, and OP_b is the lower boundary of the number of cycles required for operations inside the block.

For a given II_{sys}, the latency of the hardware accelerator for computing a layer batch $T_{batch}^{hardware}$ is:

$$T_{batch}^{hardware} = F_{out}F_{in}\Big(f_h f_w \left\lceil \frac{N_{in} \times WL_{bus}}{WL_q} \right\rceil II_{sys} + T_{communication}\Big) \tag{4}$$

where WL_{bus} denotes the width of the bus (DMA) in the platform, and WL_q denotes the wordlength of the input. $T_{communication}$ captured the required time for setting up the control parameters via AXI4-Lite and, in the case of a convolution layer, for retrieving weights and biases.

4.2 Software Latency Model

The overall latency of the system depends also on the latency introduced by the tasks executed on the ARM processor. The overall latency model has been refined to account for the above overheads leading to:

$$T_{batch} = T_{batch}^{hardware} + T_{batch}^{cpu} \tag{5}$$

where T_{batch}^{cpu} models the time spent by the CPU. The actual value of T_{batch}^{cpu} mainly depends on flushing and invalidation of the cache for DMA transfers. T_{batch}^{cpu} can be measured and estimated for each device by experiments.

The overall latency of the system, assuming N layer batches in total, is given by:

$$T_{sys} = \sum_{i=1:N} T_{batch}^{i} \tag{6}$$

where T_{batch}^{i} denotes the latency for executing batch i.

4.3 Resource Estimation

DSP utilisation: The proposed architecture utilises DSP cores for the computations that are required in the convolution and sigmoid function evaluation. The architecture unrolls fully the computations in the convolution filter resulting in the utilisation of K^2 DSPs for every $K \times K$ kernel. Finally, K takes the maximum kernel size K_c^{max}. For the evaluation of the sigmoid function in the Yolo layer, the fixed point exponential function provided by Xilinx is utilised, requiring the utilisation of 2 DSPs. Given a targeted parallelism factor of p_c and p_y, for the convolution and Yolo layers respectively, the total number of utilised DSP cores of the system under configuration $\mathbf{P}$ are:

$$DSP_{sys}(\mathbf{P}) = (K_c^{max})^2 p_{c,1} p_{c,2} + 2p_y \tag{7}$$

Memory utilisation: On-chip memory is utilised mainly as convolution weights buffer and input buffer. For the weights buffer, the storage size (in words) is the product of the number of input channels (N_{in}), the number of output channels (N_{out}), and the dimensions of kernels ($(K_c^{max})^2$). As N_{in} and N_{out} are capped at N_{max}, the weights buffer needs to have a maximum capacity of $N_{max}^2(K_c^{max})^2$ words.

The proposed architecture adopts input line buffers for sliding windows in convolution and max pooling layers. For a layer with kernels of size $K \times K$, the line buffer has K lines. K should be K_c^{max} and K_p^{max} for convolution and pooling respectively. (8) captures the overall number of the BRAMs required by the architecture.

$$BRAM_{sys}(\mathbf{P}) = \left\lceil \frac{N_{max}^2 WL_q}{BRAM_{size}} \right\rceil (K_c^{max})^2 + \left\lceil \frac{N_{max} f_w^{max} WL_q}{BRAM_{size}} \right\rceil (K_c^{max} + K_p^{max}) \tag{8}$$

f_w^{max} denotes the maximum width of the input feature map, where $BRAM_{size}$ denotes the size of the BRAM in bits. The first term of (8) captures the memory requirements for storing the kernel weights, where the second term is the necessary number of BRAMs for the input buffers.

5 Design Space Exploration

A key advantage of the proposed framework is that it can tailor the architecture of the system to the resource availability of the targeted device. Moreover, the proposed framework can be deployed to any FPGA device without having hard constraints on the targeted resource availability such as the requirement of having enough on-chip memory for storing the parameters of the whole network on-chip.

The problem of mapping YOLOv3-tiny to an FPGA device targeting latency-sensitive applications is cast as an optimisation problem as follows:

$$\begin{aligned} &\min_{\mathbf{P}} T_{sys}(\mathbf{P}) \\ &s.t.\ DSP_{sys}(\mathbf{P}) \leq DSP_{avail} \\ &\qquad BRAMs_{sys}(\mathbf{P}) \leq BRAMs_{avail} \end{aligned} \tag{9}$$

where $\mathbf{P}$ is the compile-time known parameter vector of the system.

As such, given a set of resources, the proposed framework automatically searched the parameter (i.e. configuration) space $\mathbf{P}$ with the help of analytical models, in order to identify a design point that optimises the latency of the system.

6 Wordlength Optimisation

YOLOv3-tiny only reduces the structure of the network compared to Yolov3, where the parameters of the network are left as floating point numbers. The work investigates the robustness of the network when reduced precision network parameters are utilised in order to achieve a reduction on the required resources and off-chip memory accesses.

The quantisation process is guided through the Weight loss metric $Wl(q)$, that measures the impact of quantisation on the representation of the network parameters using a quantisation process q (10).

$$Wl(q) = \sum_{i,j} \frac{(w_{i,j} - \hat{w}_{i,j})^2}{w_{i,j}^2} \tag{10}$$

$w_{i,j}$ represents the original floating point weight, where $\hat{w}_{i,j}$ represents its quantised version under quantisation process q. Two quantisation schemes that have been investigated are 8-bit and 16-bit wordlength, where half of the wordlength is allocated for the fractional part. In both schemes, linear quantisation is used. Table 1 shows the impact of each configuration to the Wl metric across convolutional layers of the network. The results led to the adoption of 16-bits wordlength for the system. Tested on COCO val5k[5], the 16-bit version achieves 30.9% mAP50, compared with 33.4% of the original floating point network, indicating a small loss in detection precision.

Table 1. Weight loss of fixed point quantisation

Layer Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wl @ 8 bit(%)	7.81	2.68	3.37	11.01	24.26	48.40	7.30	70.91	17.85	6.35	1.36	60.33	3.00
Wl @ 16 bit(%)	0.00	0.01	0.01	0.05	0.11	0.26	0.03	0.56	0.08	0.03	0.01	0.44	0.01

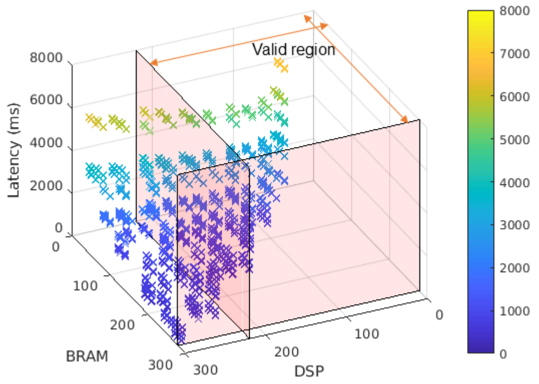

Fig. 5. Design space exploration between resources and latency, valid region is given by resources constraints

7 Evaluation

A Zedboard development kit with Xilinx XC7Z020 SoC and 512 MB DDR3 is used for the evaluation of the proposed framework. The clock frequency of programmable logic and processing system is 100 MHz and 666.7MHz respectively.

Targeting the utilisation of the whole device, the Design Space Exploration stage identifies a number of design points that meet the constraints imposed by the available resources and predicts a latency figure for each point. The traversed space is depicted in Figure 5. The best performing design achieved a latency of 532ms per inference (measured on the board) requiring 185 BRAMs, 160 DSPs, 25.9k LUTs and 46.7k FFs. The measured power consumption is 3.36W.

7.1 Performance Model Evaluation

The accuracy of the performance and resource model was investigated by producing design points under various resource constraints. The deviation on the predicted resource utilisation and achieved latency was derived by deploying the designs on the actual hardware platform. Figure 6 shows the accuracy of the derived performance models as well as the impact of including the software latency model.

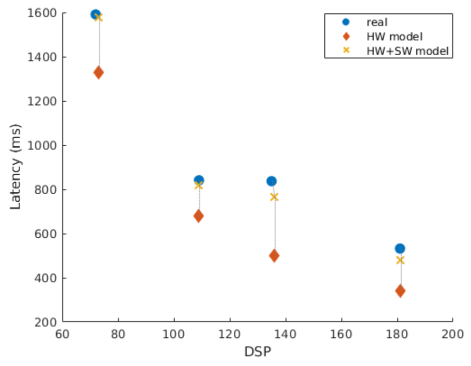

Fig. 6. Latency model evaluation across various resource targets. "real" refers to the measured performance on the board.

7.2 Comparison with CPU and GPU

Redmon et al. [14] deployed floating point YOLOv3-tiny on a Pascal Titan X achieving a 220 fps, leading to a power efficiency of 2.03 GOPS/W [11]. The proposed solution is 1.53 times more power efficient. Also, an ARM (ARM-Cortex A9 at 667MHz) based-only implementation of the system was developed and compared against the proposed solution. The results showed that the proposed system is 290 times faster.

7.3 Comparison with existing FPGA implementations

Even though FPGA implementations of YOLOv3 have not been reported in the literature, the system is positioned with work that targets the previous version of YOLO, which is a less computationally demanding model.

Table 2. Target networks comparison of the proposed design with previous works

	[18]	[12]	[9]	[10]	this work
Target Network	FPGA YOLO	Tincy YOLO	Lightweight YOLOv2	YOLOv2 tiny	YOLOv3 tiny
Data type	-	1-8b	1-32b	1-6b	16b
Test Platform	Zynq7035	XCZU3EG	ZCU102	Virtex-7 VC707	Zedboard
BRAM18k	787	-	1706	1026	185
DSP	409	-	377	168	160
LUT	47k	-	135k	86k	25.9k
FF	40k	-	370k	60k	46.7k
GOPS	-	71.04	610.93	464.7	10.45
Latency (ms)	52	63	25	9.15	532
Power (W)	7.518	6	4.5	18.29	3.36
DSE					✓

8 Conclusion

In this paper, the first latency-driven, scalable, framework for mapping YOLOv3-tiny to an FPGA device is presented. The key feature of the framework is the lifting of any assumptions on on-chip memory capacity for storing the model parameters and intermediate results, making possible the deployment of the YOLOv3-tiny object detector with limited resources. Targeting a Zedboard, the proposed system achieves a frame rate of 1.88 fps and the throughput of 10.45 GOPS. Tested on COCO val5k, a reduction in mAP of 2.5pp (30.9% vs 33.4%) is achieved under 16bit fixed point implementation without any retraining step. The presented work opens the way for low-latency object detection on low-end FPGA devices. The source code of the framework is available in github[1].

References

1. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html
2. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html

[1] https://github.com/Yu-Zhewen/Tiny_YOLO_v3_ZYNQ

3. Girshick, R.: Fast r-cnn. In: The IEEE International Conference on Computer Vision (ICCV). pp. 1440–1448 (Dec 2015)
4. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. IEEE Computer Society Conference on Computer Vision and Pattern Recognition pp. 580–587 (Jun 2014)
5. Lin, T., Maire, M., Belongie, S.J., Bourdev, L.D., Girshick, R.B., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft COCO: common objects in context. CoRR **abs/1405.0312** (2014), `http://arxiv.org/abs/1405.0312`
6. Liu, B., Xu, X.: Fclnn: A flexible framework for fast cnn prototyping on fpga with opencl and caffe. In: 2018 International Conference on Field-Programmable Technology (FPT). pp. 238–241 (Dec 2018)
7. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A.: Ssd: Single shot multibox detector. In: The 14th European Conference on Computer Vision (ECCV). pp. 21–37 (Oct 2016)
8. LogicTronix: Yolov3 tiny tutorial: Darknet to caffe to xilinx dnndk (2019), `https://logictronix.com/wp-content/uploads/2019/08/Yolov3-Tiny-Tutorial-Darknet-to-Caffe-Conversion-and-Implementation-on-Xilinx-DNNDK_August12_2019.pdf`
9. Nakahara, H., Yonekawa, H., Fujii, T., Sato, S.: A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga. In: 2018 ACM/SIGDA International Symposium. pp. 31–40 (Feb 2018)
10. Nguyen, D.T., Nguyen, T.N., Kim, H.: A high-throughput and power-efficient fpga implementation of yolo cnn for object detection. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(8), 1861–1873 (Aug 2019)
11. Nvidia: Geforce gtx titan x user guide (2014), `https://www.nvidia.com/content/geforce-gtx/GTX_TITAN_X_User_Guide.pdf`
12. Preußer, T.B., Gambardella, G., Fraser, N., Blott, M.: Inference of quantized neural networks on heterogeneous all-programmable devices. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 833–838 (Mar 2018)
13. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 779–788 (Jun 2016)
14. Redmon, J., Farhadi, A.: Yolov3: An incremental improvement (Apr 2018), `https://pjreddie.com/media/files/papers/YOLOv3.pdf`
15. Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: Towards real-time object detection with region proposal networks. IEEE Transactions on Pattern Analysis and Machine Intelligence **39** (Jun 2015)
16. Venieris, S.I., Kouris, A., Bouganis, C.S.: Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. ACM Comput. Surv. **51**(3), 56:1–56:39 (Jun 2018). https://doi.org/10.1145/3186332, `http://doi.acm.org/10.1145/3186332`
17. Wai, Y.J., bin, Z., Irwan, S., Kim, L.: Fixed point implementation of tiny-yolo-v2 using opencl on fpga. International Journal of Advanced Computer Science and Applications **9**(10) (Jan 2018)
18. Wei, G., Hou, Y., Cui, Q., Deng, G., Tao, X., Yao, Y.: Yolo accelration using fpga architecture. In: 2018 IEEE/CIC International Conference on Communications in China (ICCC). pp. 734–735 (Aug 2018)
19. Zhao, Z.Q., Zheng, P., Xu, S.T., Wu, X.: Object detection with deep learning: A review. IEEE Transactions on Neural Networks and Learning Systems pp. 1–21 (Jan 2019)