Optimizing CNN-based Object Detection Algorithms on Embedded FPGA Platforms

Ruizhe Zhao[1], Xinyu Niu[1], Yajie Wu[2], Wayne Luk[1], and Qiang Liu[3]

[1] Imperial College London
{ruizhe.zhao15,niu.xinyu10,w.luk}@imperial.ac.uk
[2] Corerain Technology
james.wu@corerain.com
[3] Tianjin University
qiangliu@tju.edu.cn

Abstract. Algorithms based on Convolutional Neural Network (CNN) have recently been applied to object detection applications, greatly improving their performance. However, many devices intended for these algorithms have limited computation resources and strict power consumption constraints, and are not suitable for algorithms designed for GPU workstations. This paper presents a novel method to optimise CNN-based object detection algorithms targeting embedded FPGA platforms. Given parameterised CNN hardware modules, an optimisation flow takes network architectures and resource constraints as input, and tunes hardware parameters with algorithm-specific information to explore the design space and achieve high performance. The evaluation shows that our design model accuracy is above 85% and, with optimised configuration, our design can achieve 49.6 times speed-up compared with software implementation.

1 Introduction

Object detection is a fundamental and difficult computer vision problem that requires the solution not only to tell what the image is about, but also to recognise the objects inside the image. A typical object detection algorithm consists of two major steps: bounding boxes regression and inner object classification. Traditional approaches like sliding window and region-based algorithms suffer from low accuracy and long execution time. Recently, several new CNN-based algorithms, which inherit successful image classification CNN architectures (e.g. VGGNet, GoogLeNet, etc.) and integrate them into object detection problem, beat old ones in accuracy (best mean average precision 83.6% on PASCAL VOC 2007 from R-FCN[3]) and in execution time (155 frames per second for Fast YOLO[9]).

While these state-of-the-art CNN-based object detection algorithms look promising, they may not be suitable to be deployed on embedded systems without modification. There are three main challenges: (1) Most of the CNN architectures for object detection algorithms do not have identical layer parameters (e.g. different convolution layers can have different kernel sizes, such as 3×3, 7×7 and 11×11), which increases the difficulty of designing generic hardware modules that can be adapted to varying parameters. (2) Object detection algorithms use deep and complex CNN architectures, which

makes it hard to fit the network into an FPGA and to decide the optimal parameters of hardware modules. (3) Multiple **backbone** CNN architectures are available to an object detection algorithm, and the more accurate an architecture can achieve, the more hardware resources it will require.

Our main contribution in this paper is a CNN accelerator design customised for object detection algorithms on an embedded FPGA platform. This design can tackle those three aforementioned challenges: (1) This design is built upon parameterised hardware modules that can be configured for different layer parameters. (2) We develop design models for estimating resource usage of deep CNN architectures. (3) We present an optimisation flow that treats two CNN-based object detection algorithms (YOLO and Faster RCNN) and their backbone CNN architectures as candidates, in order to find the optimal hardware design under different optimisation targets (e.g. speed or accuracy). At the end of this paper, we provide evaluation results for both the design model accuracy and the performance of the optimal hardware design. To the best of our knowledge, this is the first work to support end-to-end development of CNN-based object detection applications with FPGA accelerators.

2 Background and Related Work

Background. A typical CNN contains multiple computation layers which are concatenated together. There are 3 different kinds of layers that are frequently found in CNN architectures: Convolution layer (conv layer), fully-connected layer (fc layer), and max pooling layer (pooling layer). Details of these three layers are as follows.

1. **Convolution layer** mainly performs convolution operation between the input matrix - a representation for the input image or a feature map (will be discussed later), and the convolution kernel - a tiny coefficient matrix.
   Given f is the filter index, c is the channel index and C is the total number of channels, then the convolution layer can be discribed as follows:

$$O_f = \sum_{c=1}^{C} \texttt{conv}(I_c, K_{f,c}) + b_f \tag{1}$$

   This equation means that each output filter will sum up all convolution results between each channel of the input feature map (I_c) and the kernel ($K_{f,c}$). In many architectures, an activation function can be applied to the result elements, like Rectified Linear Unit (ReLU).
2. **Fully-Connected layer** is an affine transformation of the input feature vector. Fully-connected layer contains a single matrix-vector multiplication followed by a bias offset.
3. **Max-Pooling layer** performs a sub-sampling method that takes only the maximum value of each small region in the input matrix. These regions can be constructed by performing sliding window operations on the input matrix.
4. **Feature map** is the core idea to understand how CNN works. Every input and output matrix inside the CNN can be viewed as a feature map, which contains extracted features for the given image. Image classification aims at transforming

the whole feature map into object classification scores by using fully-connected layers, and object detection aims at exploring region information.

Popular CNN Architectures. There are many CNN architectures, but only a few of them have been validated on well-known datasets, and they are viewed as state-of-the-art CNN architectures. The following are some CNN architectures used in object detection algorithms. (1) **VGG16** [11] is one of the VGGNet versions with 16 convolution layers and 2 pooling layers. An appealing feature of VGGNet is that it has homogeneous kernel size (3×3) for all convolution layers, and is easy to implement on hardware accelerators. (2) **Zeiler-and-Fergus (ZFNet)** [15] is the winner of Image-Net Large-Scale Vision Recognition Challenge (ILSVRC) 2013. It is shallower than the VGGNet, and has different kernel size for different convolution layers. (3) **GoogLeNet** [14] is the winner of ILSVRC 2014. It discovers strategies to reduce the number of parameters in convolution layers, and replaces the fully-connected layers with the Average Pooling layer.

CNN-based Object Detection Algorithms. There are two CNN-based object detection algorithms discussed in this paper. One is YOLO [9], which is designed for real-time object detection; the other one is Faster RCNN [10], which extends Fast RCNN [5] with Region Proposal Network (RPN).

Both algorithms have two major components in their network architectures. The first one is the **backbone** CNN network, which is extracted from a typical CNN architecture; the second consists of extra layers that process the backbone CNN's output feature map. YOLO can choose to use GoogLeNet or a trimmed version, Faster RCNN can choose VGG16 or ZFNet as the backbone network.

Faster RCNN introduces extra layers like RoI pooling and RPN. It has been discovered that Faster RCNN is more accurate than YOLO but about 20 times slower. Deciding which algorithm to use will be introduced in the Section 5.

Related Work. There is much work related to CNN accelerator design on FPGA. Zhang et al. [16] use the roofline model and data dependencies analysis to optimise a convolution-only CNN architecture. Qiu et al. [7] successfully deploy VGGNet on an embedded FPGA platform, with several optimisation techniques like data quantisation and coefficient matrix decomposition. Chakradhar et al. present their dynamic configurable architecture among different CNN layers [2]. They also devise a compiler to work with their architecture. Farabet et al. [4] introduce NeuFlow, which is a runtime reconfigurable dataflow processor, and a compiler LuaFlow to compile high level dataflow representation to machine code. Similarly, Suda et al. [12] present a method to compile CNN configuration files into RTL code. They also introduce a systematic throughput optimisation methodology for OpenCL-based FPGA CNN accelerators [13]. In this work, we target object detection applications based on CNN algorithms, and explore the optimisation flow for various CNN backbone architectures and algorithms.

3 Architecture

This section presents the basic architecture of our hardware design, which consists of two kernels: `conv` kernel and `fc` kernel (Fig. 1). Each kernel contains an input buffer to cache data for further re-use, a computation kernel to perform convolution (`conv`) or matrix vector multiplication (`fc`), and an output buffer to store partial result before the final result is ready. Here we introduce these three components for each kernel in detail.

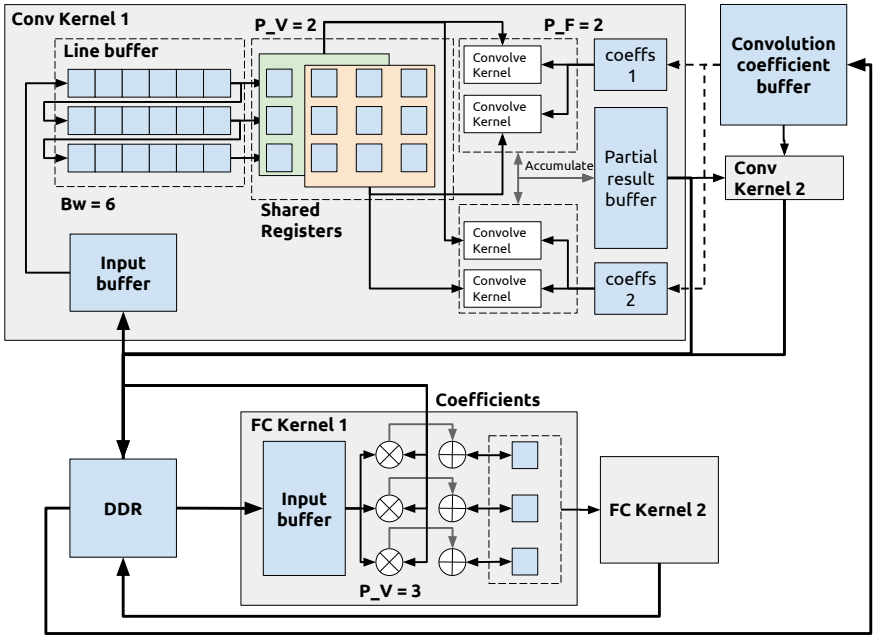

Fig. 1. A general architecture for the convolution layer (kernel size 3×3) with three different level of parallelism (P_P, P_V, and P_F). The top-left part is the line buffer.

The computation kernel inside `conv` contains several convolution kernels running in parallel, which consists of multiple multipliers followed by an adder tree. Suppose the width of a coefficient kernel is k, then the number of multipliers is k^2, and the depth of the adder tree is $\log(k)$. Multipliers take input from a customised input buffer called **line buffer** [1], which enables k data read in one clock cycle from the input feature map. The other side of the line buffer connects to a larger input buffer that partly or fully contains the input feature map. Multipliers also connect to another input buffer that caches coefficients. The output buffer in the `conv` kernel stores the partial convolution result. In each cycle the result from the adder tree will be used to update the partial result. Data type in the `conv` kernel is single-precision floating-point.

The major functionality of the `fc` kernel is to perform dot product between the re-shaped input feature vector and the coefficient matrix. The computation kernel contains

several multipliers in parallel to calculate the dot product between each row of the coefficient matrix and the feature vector. There are two ways to organise buffers: to cache the whole feature vector and store no partial output, or to store the partial result and no input buffer. These two methods are related to the computation sequence we choose for the `fc` (row major or column major), which will be discussed in Section 4. Because there is a feedback loop within the dot product, we use fixed-point data type to enhance performance. The bit width of the fixed-point data type used is 32, which contains 23 fraction bits and 8 integer bits.

4 Design Model Analysis

This section introduces the design model of `conv` and `fc`, which can predict the resource usage from given CNN architecture parameters. This design model provides an important insight into how different strategies and hardware parameters affect the usage of hardware resources, and how we could optimise performance with these model parameters. Table 1 summarises the parameters used in this paper.

Table 1. A summary of the parameters in the design model analysis

Parameter	Kernel	Description
H	`conv`	Height of the input feature map
W		Width of the input feature map
N_C		Number of channels in the input feature map
N_F		Number of filters in the output feature map
k		Height and width of the kernel
s		Stride of the convolution layer
B_H		Height of the blocked feature map
B_W		Width of the blocked feature map
M	`fc`	Length of the output feature vector
N		Length of the reshaped feature vector

The convolution layer design model takes 3 aspects into consideration. The first is **blocking**, which divides the input feature map into several parts to reduce buffer usage; the second is **data access pattern**, which is related to the exchangeable nested loops in the convolution layer. The third is computation **kernel design re-use**. Since our hardware needs to support some irregular CNN architectures with different kernel size in each layer, it is effective to re-use the same design.

Blocking Strategy. Blocking is essential when implementing `conv` kernel on FPGA. Since convolution layer's parameters are usually large in real life CNN architectures, data access patterns often cannot fit their buffer usage into the BRAM resource constraints on board. We introduce two parameters B_H and B_W to indicate the shape of the blocked input feature map. The following discussions will assume $B_H \times B_W$ blocking is applied, i.e. we will use B_H and B_W rather than H and W to indicate the input feature map's shape.

Data Access Pattern. Data access pattern is critical to `conv` kernel implementation, because we could choose to compute the convolution either by **channels** in the feature

map, or by **filters** in the output. Each of these patterns has a trade-off between the input and output buffer size.

Algorithm 1: Convolution layer computation with two nested loops.

input : A feature map I of shape $N_C \times B_H \times B_W$
input : A coefficient matrix K of shape $N_F \times N_C \times B_H \times B_W$
output: A feature map O of shape $N_F \times B_H \times B_W / s^2$

for $f \leftarrow 0$ **to** N_F **do**
  for $c \leftarrow 0$ **to** N_C **do**
    $O[f] \leftarrow O[f] + \texttt{conv}(I[c], K[f,c])$

Consider two nested loops in equation 1, one iterates the channel and the other iterates the filter (Algorithm 1). Thus we have two access patterns: **filter major** and **channel major**. The main difference between these two patterns lies in memory usage. The following will calculate the input and output buffer size. (1) **Filter major**: Algorithm 1 presents the filter major pattern. Once we complete the inner reduce add loop of channels for each output filter f in the filter major pattern, the final result for this filter will be ready. Thus, we only need to store $B_H B_W / s^2$, which is the shape of one output filter, in the output buffer. However, it needs to iterate through all the channels of the input feature map and the associated coefficient kernel, so the input buffer size of the filter major pattern is $(B_H B_W + k^2)N_C + kB_W$, where kB_W is the **line buffer** size. (2) **Channel major**: In this case, the channel iteration is the outer loop. After each iteration in the outer loop, only partial results for **all** N_F filters are available and they will be updated in the following iterations. Thus the output buffer is required to have size $N_F \times B_H B_W / s^2$. For the input buffer, only one channel of the input feature map needs to be cached, but all the coefficients for this channel should also be stored in the input buffer. Hence the input buffer size is $B_H B_W + k^2 N_F + kB_W$. The line buffer is also required for this case.

Table 2 summarises the buffer usage for these two data access patterns. With these parameterised analyses, it is convenient to decide which data access pattern should be used based on the parameter values. In general, although these two patterns have similar buffer usage, it is better to choose channel major as it has simpler control logic.

Table 2. Summary of two data access patterns

Access Pattern	Output Buffer Size	Input Buffer Size
Filter major	$B_H B_W / s^2$	$(B_H B_W + k^2)N_C + kB_W$
Channel major	$B_H B_W / s^2 \times N_F$	$B_H B_W + k^2 N_F + kB_W$

Kernel Design Reuse. According to state-of-the-art CNN-based object detection algorithms, our CNN architectures should not be restricted to VGG16, other networks like ZFNet and GoogLeNet which contain convolution layers of different kernel shapes

should also be supported in our hardware design. In order to efficiently adapt to different kernel size without re-synthesis of the design, we configure the `conv` kernel with the largest kernel size at first, and fully reuse it by adding control logic to enable computation with multiple smaller kernels. Fig. 2 illustrates how this adaptive technique works.

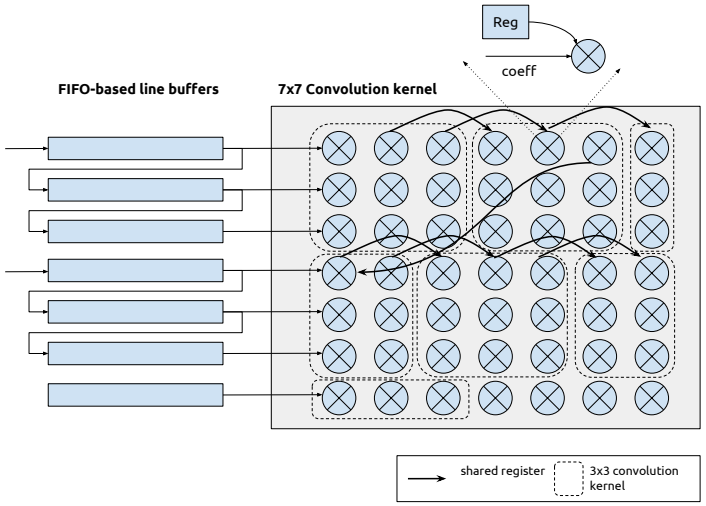

Fig. 2. Reusing 7x7 configuration for 3x3 kernel size computation. There are 49 multipliers on board, and they connect directly to the FIFO line buffer and the coefficient input port. The 7 FIFOs are split into two groups, each containing 3 FIFOs. At most, we could compute 5 3x3 kernels in parallel without reconfiguring the 7x7 kernel design. Curved arrows in the figure illustrate how the register sharing works.

A Fully-Connected layer is implemented as the `fc` kernel. As mentioned in Section 3, `fc` kernel buffer usage is mainly decided by the computation sequence, which is either **row major** or **column major**: (1) **Column major** means that we multiply all the elements in the column of the coefficient matrix to the same input value, and update the partial output with size M. (2) **Row major** requires the input vector with length N to be buffered on-chip, and reuses it to perform dot product with all the rows in the matrix. According to the discussion above, it is obvious that the computation strategy is determined by M and N: if $M \geq N$, we will use row major; and use column major when $M < N$.

5 Optimisation Flow

This section presents our optimisation flow for CNN-based object detection algorithms. The optimisation flow has three major steps: strategy selection, parameter tuning, and algorithm-specific optimisation.

Strategy Selection. Once we have the CNN network architecture configuration, we are able to select which strategy to use for each layer. There are two aforementioned strategies, one is the data access pattern for the `conv` kernel, and the other one is the computation sequence for the `fc` kernel. The selection will be based on this algorithm: For each layer i,

1. If layer i is a `conv` layer, then compare the buffer usage of all data access patterns and find the one uses minimal buffer in total.
2. If layer i is a `fc` layer, then compare M_i and N_i to decide whether to use the row major or the column major strategy.

After selecting strategies for each layer, we can derive exact expressions of the maximum BRAM usage and the maximum level of parallelisation, which are decided by both Table 2 and `fc`'s M_i and N_i.

Parameter Tuning. Suppose we are using the channel major data access pattern and row major strategy, which are suitable for most cases, we need to further tune several parameters to optimise the amount of parallelism.

1. **Pipeline depth** (P_P): For `conv` or `fc`, P_P represents the number of **kernels** to support in hardware. The supported layers can be connected as a pipeline, with the output of a layer to be the input for the next layer.
2. **Filter width** (P_F): For `conv` only, P_F represents the number of **filters** processed in parallel, which has an upper bound N_F.
3. **Vector width** (P_V): For `conv` or `fc`, P_V represents the amount of **input data** processed in parallel. While computing convolution between one kernel and one channel's feature map, it is possible to compute multiple kernels in parallel. This level of parallelisation can be measured by the width of input vector in each cycle.

Convolution Layer. Based on the above parallelism parameters, we need to modify the line buffer size, which should be $P_V B_W$ to support P_V read operations in parallel. Besides, we derive the expression for the on-chip (BW_i^{conv}) and DDR ($\overline{BW}_i^{conv}$) **bandwidth**, estimated to be:

$$BW_i^{conv} = BW_i^{out} + BW_i^{in} = P_V \times P_F + P_F \tag{2}$$

$$\overline{BW}_i^{conv} = \overline{BW}_i^{out} + \overline{BW}_i^{in} = \frac{P_V \times P_F}{F_i} + \frac{P_V \times P_F}{C_i} \tag{3}$$

It is a constrained optimisation problem to find the best P, P_F, and P_V for a given FPGA. We model resource usage of our design in two parts:

1. **Logic resources.** It covers the usage of LUT, FF, and DSP, which will linearly increase with respect to $P_P \times P_F \times P_V$ by a constant factor (L^c) decided by the layer configuration and the strategy we choose to use.

$$L^{conv} = L^c(P_P \times P_F \times P_V) \tag{4}$$

2. **Memory.** Memory usage is decided by two terms, one is buffer size (BS^{conv}), which can be calculated as follows:

$$BS_i^{conv} = (\underbrace{B_H B_W / s^2 \times N_F}_{output\ buffer} + \underbrace{B_H B_W + k^2 N_F}_{input\ buffer} + \underbrace{P_V B_W}_{line\ buffer}) \times P_P \tag{5}$$

The other is **on-chip bandwidth** (BW_i^{conv}). Buffer size decides the minimum number of BRAMs to store the data, and on-chip bandwidth decides the required number of ports as each BRAM has a limited number of ports to read and write. Thus, the memory usage for the convolution layer is:

$$M^{conv} = \max(\frac{BS_i^{conv} \times DW}{\text{BRAM}_{size}}, \frac{BW_i^{conv} \times DW}{\text{BRAM}_{bandwidth}}) \tag{6}$$

Fully-Connected Layer. The `fc` kernel can be analysed in a way similar to the `conv` kernel. As the `fc` kernel does not contain filter-wise parallelisation, there are only two parameters P_P and P_V to be decided. The **logic usage** will also linearly increase with respect to $P_P \times P_V$, and **memory size** is decided by N_i as we choose to use row major strategy. In our design, **on-chip bandwidth** for `fc` is simply $2P_V$. The **DDR bandwidth** requirement is to load coefficient data from DDR, and the input and output read and write at each cycle. Results are shown in Table 3.

Table 3. Summary of resource usage for `conv` and `fc` kernels

	`conv`	`fc`
Logic	$L^c(P_P \times P_F \times P_V)$	$L^f(P_P \times P_V)$
Memory	$\max\left(\frac{BS_i^{conv} \times DW}{\text{BRAM}_{size}}, \frac{BW_i^{conv} \times DW}{\text{BRAM}_{bw}}\right)$	$\max\left(\frac{N_i \times DW}{\text{BRAM}_{size}}, \frac{2P_V \times DW}{\text{BRAM}_{bw}}\right)$
DDR	$P_V P_F \left(\frac{1}{F_i} + \frac{1}{C_i}\right)$	$P_V \left(\frac{1}{M_i} + \frac{1}{N_i} + 1\right)$

Algorithm-Specific Optimisation. Algorithm-specific information in this context covers two algorithms: YOLO and Faster RCNN, and backbone CNN architecture candidates include VGG16, ZFNet, and GoogLeNet. At this level of optimisation, the whole application's constraints such as system capacity and real-time requirement will be taken into consideration.

Our approach is to provide two strategies: **speed priority** and **accuracy priority** for optimisation. For any object detection application, speed priority means that real-time response is important, while accuracy priority means that the estimated detection accuracy is beyond 70%. According to [9], the YOLO algorithm is suitable for speed priority, and Faster RCNN is for accuracy priority.

When we select the algorithm-specific optimisation strategy and which algorithm to use, the optimisation flow will iterate all the possible backbone CNN architectures for each algorithm, and will try to use these configurations to get the optimal result and will then compare them in order to select the best CNN architecture.

6 Evaluation

This section describes our evaluation and performance analysis of the hardware design with specific resource constraints and network architecture. We choose to measure the performance for the YOLO algorithm with the GoogLeNet backbone.

Implementation Details. We briefly introduce the implementation detail of our hardware design. We present the overall architecture in Section 3. The proposed architecture and optimisation flow can target various FPGA platforms. To illustrate our approach, our hardware design is built for the Xilinx Zynq platform (zc706), which contains two main components: PS and PL. PS is the processing system with an ARM CPU and a DDR memory, while PL refers to the FPGA, which contains logic resources, on-chip memory, and DMA support. In our case, CNN hardware design targets the PL part, with some complex software algorithms running on the PS part. We use the AXI to connect between PS and PL.

The CNN hardware design can be split into `conv` kernel and `fc` kernel. They are parameterised and are connected to each other through FIFO. They use our streaming protocol to control and schedule tasks. Coefficients and other external data will be loaded through DDR from the external memory.

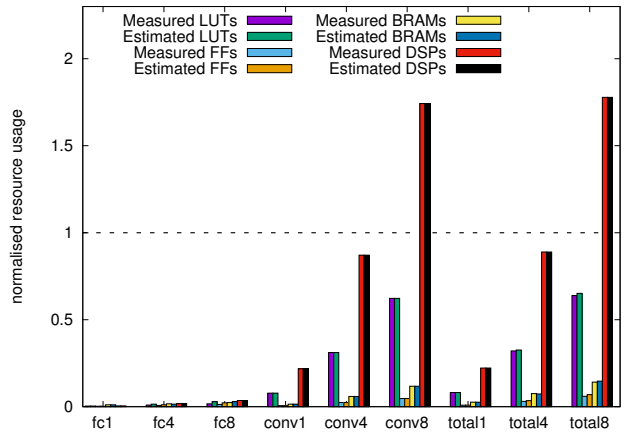

Fig. 3. Design model accuracy measured with the synthesis report and the model estimation results. Resource usage is normalised against available resources in the target chip. The last digit of each label is the P_V value.

Design Model Accuracy. We estimate the design model accuracy from the synthesis report and the estimated resource usage on 3 different cases: $P_V = 1, 4, 8$ (Fig. 3). Here the kernel size of the `conv` module is 7×7, and the column number of the `fc` module is 4096. The estimation is based on equations in Table 3. The design model accuracy is beyond 85%, and therefore it can support our optimisation flow. The dotted line stands for available resources in our target chip. Thus, we select $P_V = 4$ in this design.

Algorithm Evaluation. Based on the optimisation model, we derive the optimal design parameters for both YOLO (GoogLeNet) and Faster RCNN (VGG16), and predict the best performance for these two algorithms. In addition, we also evaluate the software performance on x86 CPU and ARM CPU. We use Darknet [8] and Caffe [6] as the software reference for YOLO and Faster RCNN evaluation. Results are listed in Table 4.

Based on the optimization model, we make a few decisions. (1) Input and output buffers are necessary so that the design has the appropriate bandwidth. (2) For the 1x1 kernel, the 25 BRAM requirement is not the major limitation in resource usage. (3) At current precision, the DSPs are the limiting resources for `conv` kernels. We can set $P_V = 4$ and $P_P = P_F = 1$ in this case. (4) `fc` kernel also uses $P_V = 4$ to coordinate with the `conv` kernel's output.

We estimate that the overall execution time for YOLO (GoogLeNet) is **0.744** second, and for Faster RCNN (VGG16) is **0.875** second. Compared with the best software performance on ARM (**36.92** seconds), the speed-up is **49.6** times. Even compared with the x86 CPU there is a 1.5 times speed-up. Although the GPU version is much faster than our implementation, the GPU (Titan X) is not suitable for embedded systems. Also the total energy cost of the FPGA version (0.868J) is much smaller than the GPU version (23J).

Table 4. Algorithm evaluation on 4 platforms

	x86 CPU	ARM CPU	FPGA	GPU
Platform	Intel Core i7	ARMv7-A	Zynq (zc706)	GeForce Titan X
Num. of Cores	8 (4 used)	2 (2 used)	-	-
Compiler	GNU GCC	GNU GCC	Vivado (2016.2)	CUDA (v7.5)
Compile Flags	`-Ofast`	`-Ofast`	-	`-Ofast`
Clock	3.07 GHz	Up to 1GHz	200 MHz	1531 MHz
Technology	45 nm	28 nm	24 nm	16 nm
YOLO (Tiny)	**1.12**s	**36.92**s	-	0.0037s (178W)
YOLO (GoogLeNet)	13.54s	430.6s	**0.744**s (1.167W)	0.010s (230W)
Faster RCNN (ZF)	2.547s	71.53s	-	0.043s (69W)
Faster RCNN (VGG16)	6.224s	Failed	0.875s (1.167W)	0.062s (81W)

7 Summary

This paper presents our novel approach to optimise CNN-based object detection algorithms on embedded FPGA platforms, which consists of a design model for the basic CNN hardware architecture, and an optimisation flow which takes into account both FPGA optimisation strategies and algorithm-specific optimisation strategies. Our

evaluation shows that an optimised hardware design for the YOLO algorithm with GoogLeNet backbone can reach 49.6 times speed-up compared with software on ARM. Also our design model accuracy is above 85%. Future work includes evaluating the object detection application with multiple real world datasets, introducing automatic data quantisation, and enhancing the optimisation flow to support CNN training.

Acknowledgement

The support of the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, UK EPSRC (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), Corerain Technologies, the State Key Laboratory of Space-Ground Integrated Information Technology, and Xilinx, Inc. is gratefully acknowledged.

References

1. Bosi, B., et al.: Reconfigurable pipelined 2-d convolvers for fast digital signal processing. IEEE Transactions on VLSI Systems 7(3), 299–308 (1999)
2. Chakradhar, S., et al.: A dynamically configurable coprocessor for convolutional neural networks. In: ISCA (2010)
3. Dai, J., et al.: R-FCN: Object detection via region-based fully convolutional networks. arXiv preprint arXiv:1605.06409 (2016)
4. Farabet, C., et al.: NeuFlow: A Runtime-Reconfigurable Dataflow Processor for Vision. In: ECVW (2011)
5. Girshick, R.: Fast R-CNN. In: ICCV (2015)
6. Jia, Y., et al.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
7. Qiu, J., et al.: Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In: FPGA (2016)
8. Redmon, J.: Darknet: Open source neural networks in c. http://pjreddie.com/darknet/ (2013–2016)
9. Redmon, J., et al.: You Only Look Once: Unified, Real-Time Object Detection (2015)
10. Ren, S., et al.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. NIPS (2015)
11. Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. ImageNet Challenge (2014)
12. Suda, N., et al.: Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA. In: FPL (2016)
13. Suda, N., et al.: Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In: FPGA (2016)
14. Szegedy, C., et al.: Going deeper with convolutions. In: CVPR (2015)
15. Zeiler, M.D., Fergus, R.: Visualizing and Understanding Convolutional Networks. In: ECCV (2014)
16. Zhang, C., et al.: Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. FPGA (2015)