On-chip Instruction Generation for Cross-Layer CNN Accelerator on FPGA

Yiming Hu, Shuang Liang, Jincheng Yu, Yu Wang, Huazhong Yang
Department of Electronic Engineering, Tsinghua University
{huym16@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn}

***Abstract*—Convolutional neural networks (CNN) are gaining popularity in the field of computer vision. CNN-based methods are computational-intensive and resource-consuming, thus are hard to be integrated into embedded systems and applied to real-time task scenarios. Many FPGA based CNN accelerators have been proposed to get higher performance.**

Cross-layer CNN accelerator is designed to reduce the data transfer by fusing several layers. However, the instruction size that needs to be transferred is usually considerable, leading to a performance drop of cross-layer accelerators. In this study, we develop an on-chip instruction generation method based on the cross-layer accelerator to reduce the total instruction size transferred to the chip. We design the corresponding hardware module and modify existing object detection models according to the hardware structure to improve the accuracy of object detection tasks.

The evaluation results show that in the same calculation process, our accelerator can achieve 35% data transfer reduction on the VGG16 network. The average instruction size and compilation time are reduced by 95% using our instruction generation method. The performance of the accelerator reaches 1414 GOP/s.

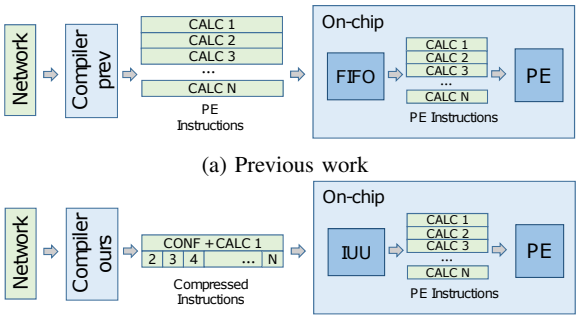
***Index Terms*—FPGA, CNN, ISA, Cross-layer, Object Detection**

I. Introduction

Convolution Neural Networks(CNNs) show great potentials in computer vision tasks and have been among the most powerful techniques for image-related tasks. However, CNN requires an enormous computation complexity, which is impossible for tasks requiring embedded or real-time processing in the real world. To deploy CNN on a mobile or low-power system, FPGA and ASIC CNN accelerators are designed to speed up the computation process.

Many cross-layer CNN accelerators have been proposed based on the idea of layer fusion [1]. The off-chip data transfer can be significantly reduced by using the cross-layer scheduling strategy. With the instruction set designs, instruction-driven cross-layer accelerators enable reconfigurable CNN computing. However, the complex scheduling strategy requires a fine-grained instruction set design, which leads to an intolerable instruction size for network inferences.

In this paper, we introduce a cross-layer FPGA accelerator with an on-chip instruction generation method for computing. To reduce the instruction size to be transferred, we propose a new *Config and Run* Instruction Set Architecture(ISA). We put the configurations of each Convolution layer into a CONF instruction. Meanwhile, we design an Instruction Update Unit(IUU), which stores the CONF instruction in an Instruction Pool(InsPool) for each layer and generates ensuing computation instructions for Processing Elements(PE) from the Compressed Calculation(C-CALC) Instruction.



(a) Previous work

(b) Our work

Fig. 1: An overview of our proposed work. (a): previous works. The calculation instructions for PE are generated by the compiler and transfered into the chip. (b): our work. The calculation instructions are highly compressed. An Instruction Update Unit(IUU) is used to generate the instructions for PEs on chip.

With the help of the proposed instruction design, the normal calculation instructions can be extremely compressed. We also implement the compiler and hardware architecture for our instruction generation method. Besides, the *Config and Run* idea of our work can also be applied to any other instruction-intensive CNN accelerators.

The contributions of this paper are as follows:

- We propose an on-chip instruction generation method for cross-layer CNN accelerators. It can also be extended to other instruction-intensive CNN accelerator structures.
- We design an Instruction Update Unit (IUU) to handle the on-chip instruction generation process.
- An object detection network is implemented on the accelerator to verify the efficiency of our work.

Experiments show that our method can significantly reduce the instruction size. The data transfer reduction on VGG-16 is reduced by 31.7%. The average instruction size and compilation time are reduced by 95%. We also implement object detection networks such as SSD [2] and YOLO [3] using the proposed accelerator. The performance reaches 1414 GOP/s. The detection Mean Average Precision(mAP) on TT100K dataset reaches 55.6 with FPS of 15.8.

978-1-7281-3391-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ISVLSI.2019.00011

The remaining part of the paper is organized as follows. Section II introduces the background and related works of CNN accelerators and object detection. Section III describes the details of our on-chip instruction generation method. Section IV shows how we implement the Instruction Updating Unit to support our instruction updating strategy. We train modified object detection models and present the experimental results in section V. Section VI concludes this paper.

II. Related Work

A. Object Detection

Convolution Neural Networks(CNNs) has been the prevailing object detection algorithm for the last several years. RCNN [4] uses CNN for classifying the proposals generated by a region proposal algorithm. Fast-RCNN [5] reuses the feature extraction layers to reduce operation numbers. Faster-RCNN [6] uses a Region Proposal Network(RPN) layer to find region proposals instead of traditional algorithms. SSD [2] and YOLO [3] generate plenty of bounding boxes and use prediction layers to refine the bounding boxes to get the detection results. [7] fuses the features from different layers to improve the detection accuracy. However, many of the techniques increase the amount of computation, which makes it difficult to meet the real-time requirements.

B. CNN Accelerator

Due to the limitation of high power consumption and low utilization rate, GPU can not be used in real-time embedded systems, motivating the research and development of many dedicated CNN accelerators. These accelerators use various strategies to map computation to hardware, exploiting different parallelism and data reuse pattern. [8] and [9] design different types of convolver to take advantage of the parallelism across kernel spatial and input/output channel dimension. Shidiannao [10] proposes an architecture using Multiply-accumulate(MAC) which involves inter-PE data flow for data reuse. [11] uses a parallel MAC and a parallel buffer array, performing at 0.26-10TOPS/W. Han [12] [13] implements a matrix-vector($M \times V$) architecture that utilizes the sparsity of neural networks.

Various works concentrate on the scheduling strategy of the CNN accelerator. [8] [9] use the intra-layer scheduling to overcome the on-chip/off-chip data transfer limitation. Alwani [1] implements a pipelined multi-layer CNN accelerator and shows that cross-layer strategy can significantly reduce the data transfer. Li [14] implements another cross-layer accelerator for AlexNet. Yu [15] proposes a flexible instruction-driven accelerator using cross-layer scheduling.

III. On-chip Instruction Generation

This section details the proposed on-chip instruction generation method. We firstly introduce the workflow of a cross-layer accelerator. Then we introduce the instruction set architecture and the instructions update mechanism.

A. Introduction to cross-layer accelerator

The main idea of cross-layer scheduling is to fuse two or more convolution layers to prevent unnecessary intermediate data transfer between the accelerator and the off-chip memory. After computing the first several rows of the layer, we can directly use the intermediate results to calculate the next layer, eliminating the data transfer for intermediate results.

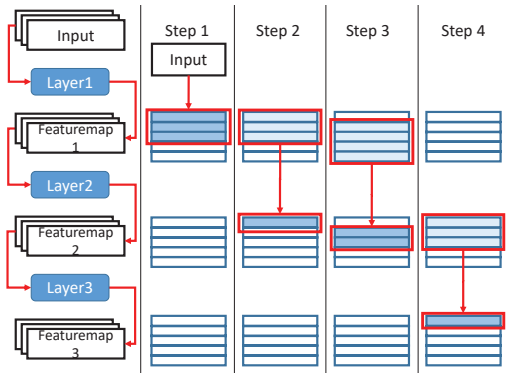


Fig. 2: An example of the cross-layer scheduling for a simple 3-layer network inference. The corresponding pseudo code is in fig. 4.

The instructions are fine-grained for a more delicate scheduling strategy. Figure 2 and fig. 4(a) illustrate the cross-layer scheduling strategy. (Step 1) The Line1-3 of fig. 4(a) calculates the first 3 lines of featuremap1 in fig. 2, (Step 2) followed by Line4 of Figure 4a, which used the first 3 lines of featuremap1 to produce the first line of featuremap2 in fig. 2. (Step 3) Line5-8 of fig. 4a repeat the cross-layer scheduling until the first 3 lines of featuremap2. (Step 4) Line9 in fig. 4 calculates the first line of featuremap3 as the input data is ready.

B. Instruction Set Overview

The advantage of cross-layer accelerators is the scheduling strategy to fuse different layers in a network. However, the previous sophisticated scheduling strategy comes at a cost: the calculation instructions must be fine-grained. Thus the number of instructions for a network inference process can be much larger than other accelerators. To solve this problem, we adopt a *Config and Run* idea for the ISA design. Firstly, we write the layer configuration to an instruction buffer called InsPool. Then the following calculation instructions can be highly compressed into the Compressed Instructions. Finally, the Instruction Generator can decode the Compressed Instructions and reconstruct the original calculation instructions.

Our proposed ISA has 4 different types of instructions, including *load data*(LOAD), *save data*(SAVE), *layer config*(CONF) and *compressed calculation*(C-CALC). The length of all instructions is 128-bit. Figure 3 depicts the instruction structures. For all instructions, the *opcode* field indicates the type and the *dep* field indicates the prerequisite operations. There is no *dep* field in C-CALC since the dependency of C-CALC instruction is solved during the compilation stage in section III-F.

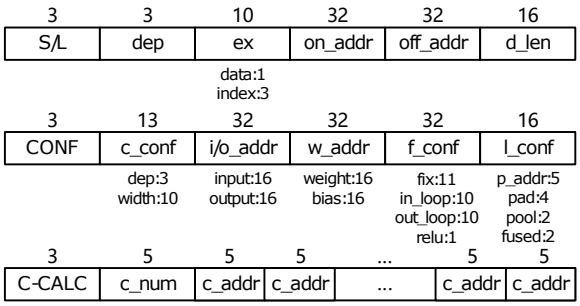


Fig. 3: The instruction structures. From top to bottom: SAVE/LOAD, CONF, C-CALC.

C. SAVE/LOAD instruction

SAVE/LOAD instructions tell the accelerator where to save/load the data or weights to/from the off-chip DDR. In addition to the *on/off-chip address* and *data length*, we set a flag to tell the instruction generator whether the data to be transferred are weights or feature maps. Since there several data/weights buffers on-chip, we use 3 bits in *ex* field to specify the target on-chip buffer for reading/writing.

D. CONF instruction

Since the PE-instructions for the same layer share some parameters such as the width of the input and output featuremaps. The other parameters of PE-instructions of the same layer can be inferred from the last-executed PE-instructions, like the read and write addresses. The CONF instruction collects the necessary shared parameters and the initial configuration of the other parameters, so that the PE-instructions can be reconstructed based on the CONF instruction.

There is exactly one CONF instruction per layer whether or not it is fused during cross-layer scheduling, and the CONF instruction should be executed before all the calculations in the same layer.

The *width* field shows the width of the current feature map, which is required for processing elements(PE) to complete the computation. The *f_conf*(feature config) field records the dynamic range of parameters for fixed-point quantization, the number of channel loops needed to be iterated, and ReLU type. The *l_conf*(layer config) field includes the InsPool address (*p_addr*) as well as other parameters of the layer including padding type and pooling type. The last 2 bits shows if the input/output feature maps are fused, which will influence the reading address during the calculation. With the help of the aforementioned instruction fields, our ISA defines all the necessary shared parameters.

The *i/o_addr* and *w_addr* shows the initial addresses of the input/output data and parameters. The addresses for the ensuing PE-instructions for the same layer can be inferred.

E. Compressed CALC instruction

Since the parameters are defined by CONF instructions or can be inferred on-chip while the model is running, we need to specify the order in which instructions are executed. Therefore, we propose the Compressed CALC(C-CALC) instruction, which stores a series of ordered InsPool addresses. Each InsPool address can be decoded into a PE-instruction.

The *c_num*(compressed number) field indicates the number of InsPool addresses in the Compressed CALC instruction, which equals to the number of PE-instructions to be calculated. The *body* field stores the InsPool addresses in sequence.

During the inference calculation, the controller sequentially reads the InsPool address and fetches the basic CONF instruction from the InsPool according to the address. Then the IUU generates the corresponding PE instruction based on the CONF instruction and sends it to PE. The maximum number of PE calculations that can be compressed in a single C-CALC instruction is :

$$\frac{L_{inst} - 8}{L_{p\_addr}}$$

where L_{inst} is instruction length and L_{p_addr} is the bitwidth of the *p_addr*. In our design, the instruction length is 128 bits, and the L_{p_addr} is 5. Thus the C-CALC instruction's efficiency has $24\times$ speedup compared to normal CALC instruction.

In the cross-layer accelerator, the most significant idea is the scheduling strategy of computation. Our *config and run* ISA keeps the shared parameters of network layers on-chip while incrementally updating the PE calculation instructions. So the redundancy of the instructions is minimized. Experiment results in section V-B show the significant improvements from our approach.

F. Compiler

We also implement the compiler for our ISA design. The compiler would arrange the CNN model into several fuse-layer subnets, solve the data dependency between instructions, output the sequences of InsPool addresses, and finally generate the addresses for LOAD, SAVE and CONF for each layer.

We use a decoder to evaluate the generated instructions. For any cross-layers scheduling strategies, the PE-instructions can be completely recovered from the compressed instructions output by the compiler. So our design will not sacrifice any scheduling space. Compared with the previous works [8], [15], the compiler for out ISA does not generate the addresses and other layer parameters for C-CALC instructions, which consume most of the compiling time, thus making the compilation process much faster.

G. Code Example

Figure 4 gives a simple code example with previous PE-Instruction set and our proposed Compressed-Instruction set, and this simple example shows that the Compressed-Instruction set can efficiently reduce the code length for cross layer. *"CALC Layer1 Row3"* indicates the calculation of the 3^{rd} row of Layer1. *"CONF Layer1"* indicates the configuration of Layer1 with the first calculation of Layer1 Row1. *"C_CALC 1 2 1 2"* means the layer index order in cross-layer scheduling, and the order is the same with the PE-Instructions.

```
01 CALC Layer1 Row1
02 CALC Layer1 Row2
03 CALC Layer1 Row3
04 CALC Layer2 Row1
05 CALC Layer1 Row4
06 CALC Layer2 Row2
07 CALC Layer1 Row5
08 CALC Layer2 Row3
09 CALC Layer3 Row1
```

(a) PE-Instructions

```
01 CONF Layer1
02 C_CALC 1 1

03 CONF Layer2
04 C_CALC 1 2 1 2


05 CONF Layer3
```

(b) Compressed-Instructions

Fig. 4: Code examples without/with our Compressed-Instruction Set.

IV. Instruction Update Unit

We design an Instruction Update Unit(IUU) to process the CONF instruction and the C-CALC instruction. The IUU mainly includes two parts: the Instruction Pool and the Instruction Generator.

Figure 5 depicts the main structure of the IUU. When a Compressed Instruction arrives, an instruction decoder firstly decides the type of the instruction according to the *opcode*. IUU will directly pass the SAVE/LOAD instructions to the Data Mover to complete the data reading/writing process. For CONF instructions, the IUU initializes the shared and initial parameters in the target InsPool address which is set by *c˙addr*) field. For C-CALC instructions, the IUU fetches the basic configuration instruction from the InstPool and sends it to the Instruction Generator. The Instruction Generator generates the calculation PE instruction which can be executed on the PEs. Besides, Instruction Generator also updates the configuration instruction and write it back to InsPool.

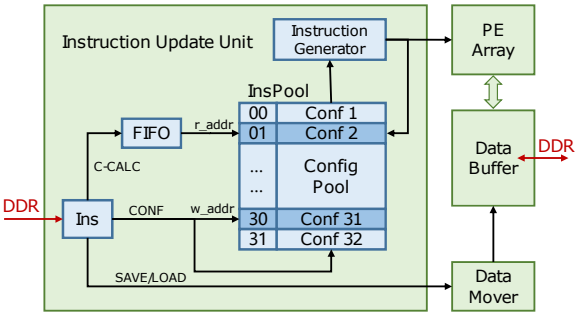


Fig. 5: An overview of the Instruction Update Unit's work flow.

A. Instruction Pool

The Instruction Pool(InsPool) is a small buffer with a size of 32×128-bit. Therefore, we can store a max of 32 CONF instructions simultaneously on-chip, supporting up to a 32-layer cross-layer scheduling inference. Though cross-layer scheduling eliminates intermediate data transfer, the weight buffer size also limits the number of fused layers, so we find that 32 is enough for all of our experiments in section V.

When a CONF instruction arrives, IUU reads the *c_addr* field and stores the config instruction to the corresponding address in the Instruction Pool. The CONF instruction contains the initial address of the data and weights needed by calculation as well as the necessary arguments of the current layer. The initialized config will be immediately sent to the Instruction Generator to proceed the first calculation.

For C-CALC instructions, IUU reads all the *c_addr* sequentially, fetches the corresponding configs from the instruction pool and passes it to the Instruction Generator. Then the InsPool will receive the configs for the next calculation from the generator and updates the configs to the same address of InsPool.

B. Instruction Generator

The Instruction Generator outputs the PE Instruction, which can be directly executed on the PE array. Besides generating PE Instructions, IUU also initializes the registers storing the cyclically varying variables, such as when to write the result back(which should be set *true* when completing all of the input channels for specific output channels) and index of the target data buffer, etc.

Figure 6 shows the structure of an Instruction Generator. It includes a Status Pool, which consists of 32 pairs of counters. Each pair of counters records the number of input channels and output channels that have already calculated for current layer. Given an InsPool request, the Instruction Generator outputs the PE-instruction and updates the internal status to the corresponding Status Pool.

During every generation process, the Instruction Generator updates the received CONF instruction. The shared parameters remain unchanged and modification to other fields of the instruction are controlled by the status in the Status Pool. We use different colors to describe the relationship in fig. 6. For example, whenever the counter CTR_i is triggered, the *o_addr_offset* will be added to the *o_addr* field of the output new CONF instruction.

When receiving a CONF instruction, the Instruction Generator would initialize the corresponding Status Pool, and simultaneously outputs the first PE-instruction for the newly configured layer.

V. Experiments

Experiments are carried out on the architecture introduced in section III with the IUU in section IV. We explore multiple network structures in this section and deploy several networks on the FPGA.

A. Experiment Setup

a) FPGA: We evaluate our work on Xilinx KCU1500 with KU115 FPGA [16]. The default value of input parallelism P_i and output parallelism P_o is set to 4. We also evaluate the performance with other values in section V-B. The maximum bitwidth is 14 for the weight buffer and 13 for the data buffer. So the sizes of the weight/data buffer are 2MB/1MB respectively. The controller reads a 128-bit instruction at a time, just like [15] does.

TABLE I: Data Transfer Comparison with On-chip Instruction Generation

Model	Feature Transfer[1] (MB)	Weight Transfer (MB)	#Instruction		Instruction Size(MB)		Total Transfer(MB)		Transfer Reduced Rate	
			[15]	Ours	[15]	Ours	[15]	Ours	Ins	Total
YOLOv2_224	2.89	57.3	1901207	85689	14.51	0.65	74.7	60.84	95.52%	18.55%
YOLOv2_448	9.65	57.3	3431143	149769	26.18	1.14	93.13	68.09	95.65%	26.89%
VGG-11_224	1.7	8.86	805881	35803	6.15	0.27	16.71	10.83	95.61%	35.19%
VGG-13_224	3.78	8.97	863291	38355	6.59	0.29	19.34	13.04	95.60%	32.57%
VGG-16_224	6.34	14.03	1322750	58646	10.09	0.45	30.46	20.82	95.54%	31.65%
VGG-19_224	8.75	19.09	1781441	78169	13.59	0.6	41.43	28.44	95.58%	31.35%

[1] Feature maps are counted twice for reading from and writing to external memory, except the input image and the final result.

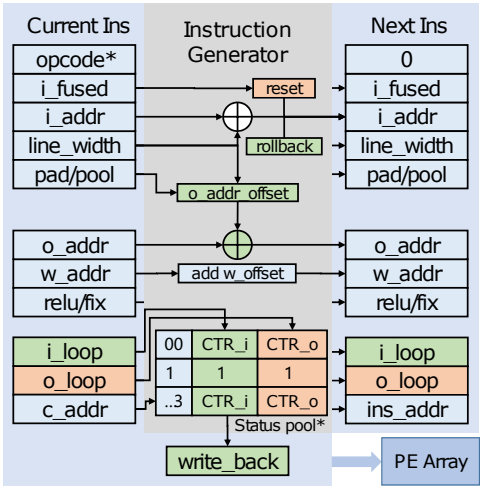


Fig. 6: The structure of the Instruction Generator. Note that whenever a new CONF instruction arrives, the related counters in the status pool will be reset.

TABLE II: Data Transfer Reduction Rate with Different P_i, P_o

Input shape	$P_i = P_o = 8$	$P_i = P_o = 4$
224	10.5	31.65
512	10.48	31.53

b) Dataset: We choose VOC [17] and TT100K [18] as the dataset for training and evaluation. The VOC object detection data set has 20 object categories, and are divided into training, verification and testing sets. An official evaluation code is also provided. It has been widely used to evaluate an object detection algorithm. TT100K is a great traffic-sign benchmark from 100000 Tencent Street View panoramas. The image resolution in TT100K is 2048×2048. The height and width of traffic signs in the photos range from less than 10 pixels to several hundred pixels. Thus the average relative size of the boxes to the full images is much smaller than the other datasets.

We apply the Fixed-point Quantization to the networks for on-chip computing. Besides, we also introduce several optimizations to the existing object detection algorithm to improve the accuracy and efficiency for small object detection tasks, such as Intermediate Feature Upsampling and Prediction Branch Pruning. Intermediate Feature Upsampling double upscales the feature maps in the middle of a forward pass and then continue the remaining layers. Prediction Branch Pruning removes the last few prediction branches which are usually for large objects. We adopt the Fixed-point Quantization in [8] to produce a hardware-friendly fixed-point CNN model.

B. Instruction Set Optimization

We implement a compiler for our ISA and evaluate the instruction size on VGG-A, VGG-B, VGG-D, VGG-E [19], and YOLO [3]. Table I lists the instruction size and the reduction of data transfer with/without on-chip instruction generation. Since the scheduling strategy does not affect the total number of CALC instructions, we compile the instructions with the same scheduling strategy for all networks: we only fuse the first five layers of the network.

The evaluation results show that we reduce the instruction size by an average of 95% compared with previous instruction-driven cross-layer accelerator [15]. For the widely used VGG-16 network, the instruction size is reduced from 10.09MB to 0.45MB. As is shown in the table, the total data transfer is also reduced by 32%.

We tried different value of input size and P_i,P_o. Table II shows the comparison with VGGNet-16. We can see that when P_i and P_o become smaller, the data transfer has a greater reduction rate. For the hardware platform with less available computing resources, the maximum parallel of the calculation is limited, which leads to a large number of instructions to be executed for network inference since PE can only complete a P_i to P_o channel computation per instruction. By applying the on-chip instruction generation technique, the required transfer bandwidth for instruction is greatly reduced.

Compared to the previous works [8], [15], our compilation without redundant address computing is more efficient and faster. The compilation time of PE-instructions for a VGG-16 network is 62s. We achieve 2.25s with a $\times 27.6$ times speedup for the compilation stage.

C. Object Detection

We deploy YOLO [3] and SSD [2] on our hardware. The evaluation results are shown in Table III. All models are modified from the original one to fit the hardware constraints of our accelerator. Figure 7 depicts the example detection results on TT100K and VOC2007 datasets.

TABLE III: Performance of Deployed Object Detection Algorithm

Model	Dataset	Acc	Fixed-point Acc	#Param(M) (M)	Complexity (GFLOPS)	FPS	Performance(GOP/s)	
							[15]	Ours
YOLO	VOC	62.3	61.8	14.3	30.6	33.1	942	1013
SSD	VOC	73.1	73	20	60.8	18.9	1150	1365
SSD	TT100K	54.1	73	20	60.8	18.9	1150	1365
SSD[1]	TT100K	56.2	55.4	20	89.3	15.8	-	1414

[1] Optimized according to section V-A

(a) TT100K (b) VOC

Fig. 7: Example Detection Results. The image of TT100K result is cropped to show the boxes clearly.

VI. Conclusion

In this paper, we propose an on-chip instruction generation method to optimize the data transfer for cross-layer CNN accelerators. An ISA is presented to compress the continuous calculation instructions with similar structures. We also design an instruction update unit module with on-chip instruction buffer and instruction generator to support performing the instruction update process. We modify and quantize some existing object detection CNN models and deploy them on the FPGA platform, achieving a 1.2× speedup compared to the previous cross-layer accelerator.

Acknowledgement

The authors gratefully acknowledge the support from TOYOTA, Xilinx and Beijing Innovation Center for Future Chips. This work was supported by National Key R&D Program of China 2018YFB0105005, National Natural Science Foundation of China(No. 61622403, 61621091), the project of Tsinghua University and Toyota Joint Research Center for AI Technology of Automated Vehicle(TT2018-01).

References

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct. 2016.

[2] W. Liu, D. Anguelov, and A. C. others, "SSD: Single Shot MultiBox Detector," in *Computer Vision, ECCV 2016*, Lecture Notes in Computer Science, pp. 21–37, Springer, Cham, Oct. 2016.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788, 2016.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

[5] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448, 2015.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, pp. 1137–1149, June 2017.

[7] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *CVPR*, vol. 1, p. 4, 2017.

[8] J. Qiu, J. Wang, S. Yao, K. Guo, *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pp. 26–35, ACM, 2016.

[9] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 127–138, Jan. 2017.

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[11] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 Envision: A 0.26-to-10tops/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 246–247, Feb. 2017.

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 243–254, IEEE Press, 2016. 00204.

[13] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 75–84, ACM, 2017. 00025.

[14] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pp. 1–9, IEEE, 2016.

[15] J. Yu, G. Ge, Y. Hu, X. Ning, J. Qiu, K. Guo, Y. Wang, and H. Yang, "Instruction Driven Cross-layer CNN Accelerator for Fast Detection on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, pp. 22:1–22:23, Dec. 2018.

[16] "Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit," 2019.

[17] M. Everingham, S. M. A. Eslami, *et al.*, "The Pascal Visual Object Classes Challenge: A Retrospective," *International Journal of Computer Vision*, vol. 111, pp. 98–136, Jan. 2015.

[18] Z. Zhu, D. Liang, S. Zhang, X. Huang, B. Li, and S. Hu, "Traffic-sign detection and classification in the wild," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2110–2118, 2016.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.