Reconfigurable Hardware Generation for Tensor Flow Models of CNN Algorithms on a Heterogeneous Acceleration Platform

Jiajun Gao[1], Yongxin Zhu[1,2](✉), Meikang Qiu[3], Kuen Hung Tsoi[4], Xinyu Niu[4], Wayne Luk[5], Ruizhe Zhao[5], Zhiqiang Que[5], Wei Mao[6], Can Feng[6], Xiaowen Zha[6], Guobao Deng[6], Jiayi Chen[6], and Tao Liu[6]

[1] School of Microelectronics, Shanghai Jiao Tong University, Shanghai, China
zhuyongxin@sari.ac.cn
[2] Shanghai Advanced Research Institute, Chinese Academy of Sciences, Shanghai, China
[3] Harrisburg University of Science and Technology, Harrisburg 17101, PA, USA
[4] Shenzhen Corerain Technologies Co. Ltd., Shenzhen, China
[5] Imperial College London, London, UK
[6] The Commercial Aircraft Corporation of China, Shanghai, China

Abstract. Convolutional Neural Networks (CNNs) have been used to improve the state-of-art in many fields such as object detection, image classification and segmentation. With their high computation and storage complexity, CNNs are good candidates for hardware acceleration with FPGA (Field Programmable Gate Array) technology. However, much FPGA design experience is needed to develop such hardware acceleration. This paper proposes a novel tool for design automation of FPGA-based CNN accelerator to reduce the development effort. Based on the Rainman hardware architecture and parameterized FPGA modules from Corerain Technology, we introduce a design tool to allow application developers to implement their specified CNN models into FPGA. Our tool supports model files generated by TensorFlow and produces the required control flow and data layout to simplify the procedure of mapping diverse CNN models into FPGA technology. A real-time face-detection design based on the SSD algorithm is adopted to evaluate the proposed approach. This design, using 16-bit quantization, can support up to 15 frames per second for 256*256*3 images, with power consumption of only 4.6 W.

Keywords: FPGA · Framework · CNNs · Hardware acceleration

1 Introduction

In the era of big data, massive data is collected in people's everyday life. How to extract high semantic information and conduct efficient data analysis from these raw data is always a hot topic recently. Convolutional Neural Networks (CNNs) [1] based algorithms have achieved great performance and high accuracy in many applications related to computer vision, such as object detection [2] image segmentation [3] and speech recognition [4]. State-of-the-art CNN-based object detection algorithms like SSD [5], YOLO [6], etc. have been applied to realistic applications and can reach near-human accuracy.

However, the CNN algorithms are very computationally intensive which becomes a major issue in their application to real time tasks on embedded devices. Due to their highly-parallel and bit-oriented architecture, FPGAs have been widely adopted to accelerate these algorithms. According to survey [7], FPGA-based accelerators achieve higher performance in terms of execution time compared with CPUs, consume much less power than GPUs, and tend to be more flexible and configurable than ASICs.

FPGAs can provide high performance for specified network topology at a time through off-line reconfiguration. To implement one CNN model with FPGA, designers should understand the network topology and the flow control with FPGA modules. It is not friendly to the developers who focus on high level machine learning models or neural network architectures. Moreover, off-line reconfiguration also takes considerable efforts and add to complexity of application development [8]. To make FPGAs accessible to a broad community of CNN application developers who are versed in CNN algorithms but lack hardware design experience, we provide a design tool, CNNBUILDER, to help deal with the challenge. Our main contribution in this paper is a reconfigurable hardware generation tool for CNN algorithms targeting a heterogeneous acceleration platform and we make our contributions as follows:

(1) We propose a design tool, CNNBUILDER, which adopts a unified structure to cover different CNN models and save them locally as model description files. This enables our approach to support a high-level programming interface adopted by TensorFlow.
(2) To enable automation of flow control and FPGA re-configuration, we adopt a directed graph structure to describe a design in a model description file.
(3) A memory management facility has been developed to automate memory address allocation to adaptively generate data layout to make the most effective use of limited on-chip resources.

This paper aims to make energy-efficient FPGA accelerator easy to use, and to extend the versatility and improve designer productivity in project development. The rest of the paper is organized as follows: Sect. 2 introduces related work on mapping high-level neural network models to FPGAs. Section 3 introduces relevant CNNs and FPGA accelerator architectures. Section 4 presents our proposed framework design, including unified data structure design, memory allocator design and flow control design. Section 5 provides evaluation result with SSD model to show the improvement in performance and productivity.

2 Related Work

There exists some similar work in this area on mapping high-level neural network models to FPGAs. Sharma et al. [9] devised a design tool DNNWEAVER that automatically generates a synthesizable accelerator for given (DNN, FPGA) pair from a high-level specification in Caffe. Wang et al. proposed a framework DeepBurning [10] to simplify the procedure of mapping diverse neural networks into FPGAs.

Compared with the above two frameworks, our approach exposes a high-level programming interface based on TensorFlow model files instead of Caffe in [9] and [10]. Secondly, similar to DNNWEAVER [9] and DeepBurning [10], our design tool covers multiple neural network models and maps them into FPGA. However, our approach follows a streaming architecture and does not involve instructions, while DNNWEAVER adopts an instruction set architecture. Lastly, DNNWEAVER [9] and DeepBurning [10] only support FPGA implementations, while our approach supports both CPU and FPGA technologies. The device type can be configured through script files, which allows easy realization of heterogeneous acceleration.

3 Background

3.1 CNN

A typical CNN model always contains an input and an output layer, as well as multiple hidden layers. The most frequently used layers in CNN are: convolutional layer, pooling layer and activation layer. The CNN algorithm we implement with CNNBUILDER in this paper is SSD [5], and its structure is shown in Fig. 1. SSD's architecture builds on the VGG-16 architecture but discards the fully connected layers. VGG-16 is used to extract feature maps and after that SSD applies 3*3 convolution filters for each cell to make predictions.

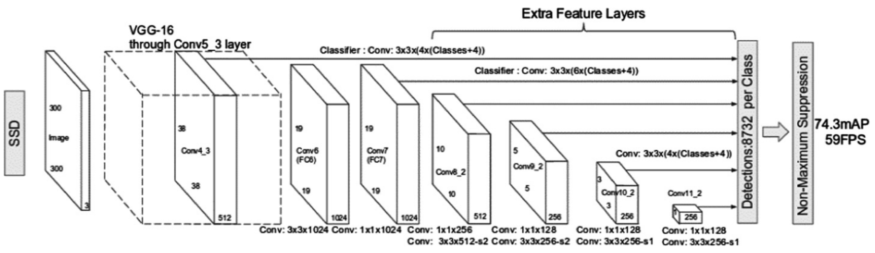


Fig. 1. SSD architecture

3.2 FPGA Accelerator Architecture

The architecture of our proposed FPGA accelerator, shown in Fig. 2, takes several components into consideration, including computation units design (mainly convolution computation unit), on-chip memory, external memory and interaction between the on-chip data and the off-chip data. Convolution is the most important unit for CNN-based algorithm which convolves the input feature maps with the convolutional kernel and produces the output feature map. Because of the on-chip memory resource constraint, it is hard to fit the entire CNN into FPGA board. So, all data for processing are stored in external memory first, and then cached in on-chip buffers layer by layer before they are processed by computation units. In addition, there is also an AXILite Bus which is responsible for the control logic between FPGA Program logic (PL) and the processor (PS). The convolution computation unit is mainly based on design in [2].

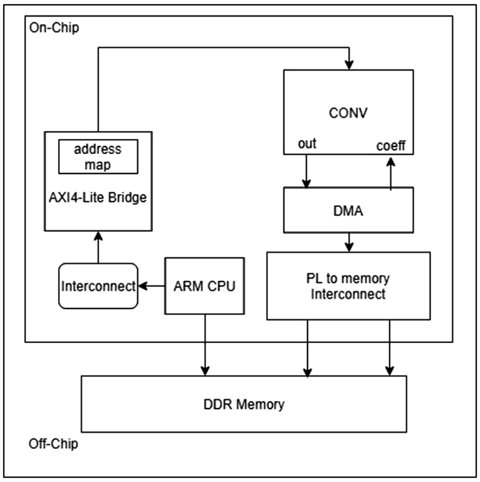


Fig. 2. FPGA accelerator architecture

4 Design Tool Description

4.1 Overview

Figure 3 depicts the overall architecture of our approach. We define three kind of files:

(1) **Model description file.** A unified structure, which will be described in the next section, is adopted to support different CNN models. This file contains the essential information of the computation dataflow graph of a specific CNN model.
(2) **Coefficient data file.** The weight parameters of each layer will be captured as binary files with layer name. These files will be loaded into an FPGA afterwards.
(3) **Data layout configuration file.** This file is used to describe the size of input feature map and output feature map. With this file, our tool can pre-allocate memory space automatically.

As depicted in Fig. 3, CNNBUILDER automatically transforms the programmer-provided CNN model files generated by a TensorFlow platform to model description file, coefficient files and data layout configuration files. Then, with these files, our design tool maps CNN models into an FPGA. In this way, developers can start with TensorFlow and training models. Our tool can then be used to produce an FPGA implementation from TensorFlow descriptions.

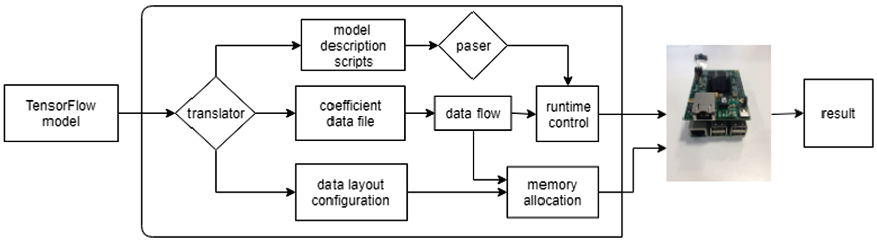


Fig. 3. Framework of CNNBUILDER

4.2 Unified Data Structure Design

Our approach provides a unified data structure to abstract away the details of FPGA accelerator design. We use Google's protocol buffer to design the structure and abstract different kinds of CNN models into this structure. We define **DFGNode** to capture different layers in CNN models and it contains the following information:

- **Name** is the name of the node, which is always the same as the corresponding layer in the model. It is the unique ID to identify the node.
- **Input** represents the input node or input nodes of current node.
- **Operation** represents the operation to be executed in this node. The operations CNNBUILDER supports now are: convolution, max-pooling, fully-connected.
- **Device Type** can be set to be either FPGA or CPU, which decides whether to use FPGA accelerator for current node.
- **Data Type** can be set as FLOAT or FIXED32 or FIXED16, which corresponds to the device type. If the device type is set as CPU, data type will be FLOAT. If the device type is set as FPGA, data type should be FIXED32 or FIXED16.
- **Operation parameter** contains necessary information of operation parameters and every node contains one operation parameter.

We have defined different operations to support different CNN models. Details of these operations are shown as follows:

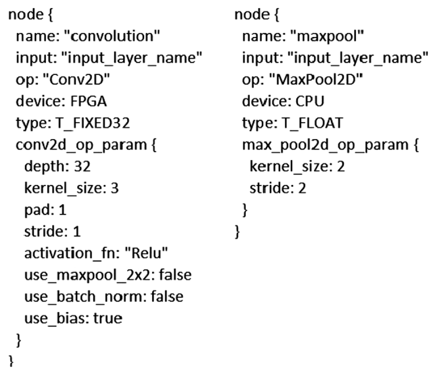
Input for input node: There is only one node with input operation in specified CNN model. It contains dimension information of the input feature map.

Conv2D for convolution: device type can be FPGA or CPU to operate on different platforms. As mentioned in FPGA accelerator architecture, we have designed adding bias, activation, pooling (max-pooling2*2 in this paper), and batch normalization in convolution module and set some signals to activate corresponding functions. Activation in table can be ReLU, Tanh, Sigmoid, and the default value for activation param means no activation function is used.

MaxPool2D for max-pooling: This node is designed for CPU platform. We subsample each 2*2 window of input feature map to a single maximum pooled output. The height and width of the window are fixed to 2.

FullyConnected for fully connected: Fully connected layer is implemented as matrix multiplication between weight matrix with dimensions (rows * columns) and input matrix with dimensions.

There are some operations with no parameters, such as **Drop-out**. We use the structure to store the information of the model. Figure 4 shows a convolution node and a max-pooling node in model description script.



```
node {
 name: "convolution"
 input: "input_layer_name"
 op: "Conv2D"
 device: FPGA
 type: T_FIXED32
 conv2d_op_param {
  depth: 32
  kernel_size: 3
  pad: 1
  stride: 1
  activation_fn: "Relu"
  use_maxpool_2x2: false
  use_batch_norm: false
  use_bias: true
 }
}
```

```
node {
 name: "maxpool"
 input: "input_layer_name"
 op: "MaxPool2D"
 device: CPU
 type: T_FLOAT
 max_pool2d_op_param {
  kernel_size: 2
  stride: 2
 }
}
```

Fig. 4. Conv2D and MaxPool2D in model description file

4.3 Memory Allocator Design

This part will elaborate the design of memory management interface to allocate memory automatically. Besides model description file, we also extract coefficient data files and data layout configuration files. Coefficient data files contain the parameters of each layer. And the data layout configuration files include the size of input feature map and output feature map, as well as the shape of the coefficient tensors.

The memory allocator is based on a Best-fit with Coalescing algorithm with basic functions including memory allocation, release, and fragment management. The idea behind this algorithm is to divide the memory into a series of memory blocks, each of which is managed by a block data structure. From the block structure, information such as the base address of the memory block, the usage state of the memory block, the block size, the pointer to the previous and the next block can be obtained. The entire memory can be represented by a block structure with a double-linked list as shown in Fig. 5.

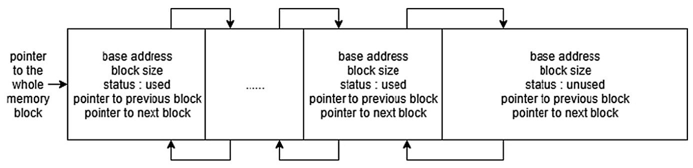


Fig. 5. Framework of memory allocator

The information of the shapes of the input feature map, the weight data and the intermediate feature map is included in the data layout configuration files. Based on layout configuration files, a memory allocator will fetch and store data sets to pre-allocated addresses. It will allocate memory of the same size depicted in the configuration files for each layer. After that, input feature map and weight data will be loaded to specified address in the tool at first and then be loaded into on-chip buffers when related operation are performed. Our approach makes use of pairs of layer name and base address to save information for flow control, which will be covered in the next section.

4.4 Automatic Parser and Runtime Control Design

For each model description file generated with unified data structure, we design a parser to map the specified CNN model into FPGA. For these **DFGNode**s except input node, we can use directed edges with input node as source and current node as target to construct a **Data Flow Graph** (**DFG**). **DFG** is a class that contains essential information of the computation dataflow graph of a specific CNN model. With **DFG**, we realize runtime control as shown in Fig. 6.

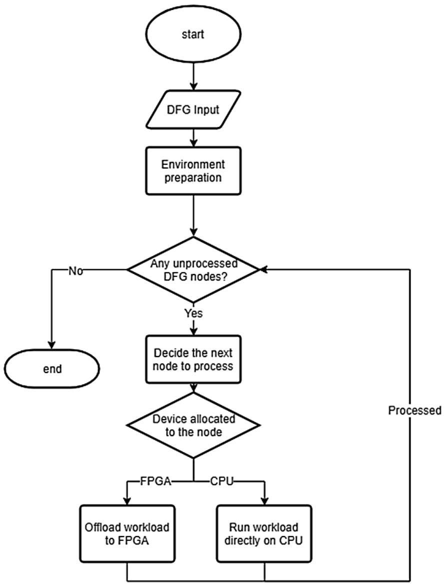

start
DFG Input
Environment preparation
Any unprocessed DFG nodes?
No
Yes
end
Decide the next node to process
Device allocated to the node
FPGA
CPU
Offload workload to FPGA
Run workload directly on CPU
Processed

Fig. 6. Runtime control flowchart

Firstly, memory allocator will conduct environment preparation. Then, our design tool searches for the input node in **DFG** and find the starting address of the corresponding data by the name of current node. After that, it loads these data into an on-chip buffer through DMA. It will detect the status register of DMA until the end of the

DMA transfer. It fetches the information of the node and sets related signals by writing registers including starting signal of computation. The intermediate output feature map will be dumped to specified address through DMA and saved as input of next node. For any unprocessed **DFGNode**, our tool repeats the process until all nodes in DFG are traversed. After that, it returns pairs of the name of output node and the starting address of the corresponding data.

5 Evaluation

5.1 Implementation Details

The FPGA based accelerator is provided by Shenzhen Corerain Technology. It is built on a Xilinx Zynq ZC706 board which consists of a Xilinx XC7Z045 FPGA, a dual ARM Cortex-A9 Processor and 1 GB DDR3 memory. The FPGA XC7Z045 is programmed with the convolutional neural network accelerator mentioned in this paper. The ARM processor is used to initialize the accelerator and run our design tool. All designs run on a single 150 MHz clock frequency and the DDR3 memory has a datapath width of 64 bits. The ARM core reorganizes the input feature map and coefficient data, and then stores them to specified address generated by memory allocator described in the previous section. The FPGA accelerator accesses the DRAM memory through AXI switches.

Our design tool aims to map trained CNN model into FPGA and focuses on the inference instead of training models. Since it differs in FPGA platforms and the design of FPGA accelerator compared with prior work, it is hard to compare the proposed design tool with them directly. Here is the evaluation method in this paper: an experienced engineer knows deep learning and FPGA accelerator design well from Corerain Technology write the code to drive FPGA manually and the time used will be compared with the corresponding design in our approach. Meanwhile, the accuracy and power consumption will also be evaluated.

Application. In this paper, our design tool maps trained SSD5 model into FPGA. In order to be better implemented on the FPGA, the SSD5 model is adjusted with input size of 256*256*3.

In experiments, we map the well-trained SSD model onto FPGA with our design tool and records the time it takes to complete a round of network forward-propagation with the input set. We are going to compare the performance of using and not using CNNBUILDER. Function correctness is based on FDDB (Face Detection Data Set and Benchmark) [11] to evaluate the accuracy.

5.2 Experimental Results

Performance and Power, we use FDDB [11] as input and record the run time it takes to process the feature maps with and without our tool targeting FPGA design. Our tool can support CPU as well and we also record the time taken on CPU platform. The results are shown in Table 1. *MC* represents manually-coded driver for the application

and it is a reference for our design tool in our experiments, which is denoted as *AG* (Automatic Generation).
Compared with manually coded implementation, automatic generated drivers from our approach contains more software operations which leads to extra time consumption. As shown in Table 1, the average convolution time using our tool is 150 ms, which is very close to 142 ms with manually coded implementation. In manually coded implementation, the lines of code to be handwritten is nearly a thousand and for each CNN model, these implementations need to be modified manually.

Table 1. Our design tool with FPGA and CPU implementations

Device	FPGA	CPU
Platform	ZC706	Intel Core i5
Compiler	Vivado	GCC (4 cores)
Clock	150 MHz	2.30 GHz
Precision	32-bit fixed-point	32-bit floating-point
MC Conv. time	142 ms	366 ms
AG Conv. time by CNNBUILDER	150 ms	366 ms
MC power	4.3 W	-
AG power by CNNBUILDER	4.6 W	-

FPGA's power consumption is obtained from the board using a power meter. With no program running, the power consumption of the FPGA board is 3.6 w. When implementing the SSD algorithm, the power consumption of designs developed with our tool is 4.6 w, only 1.07 times of that with manually-coded driver.

Accuracy. In this experiment, FDDB [11] is used to evaluate the functional correctness and the accuracy of position coordinates and size of face detection frame with our design tool. The result of manually coded implementation and our design tool is the same and the true positive rate reaches up to 82.76% in the case of a false positive number of 50. The result of golden-reference application implemented with SSD model is 82.92% with the same false positive number. Considering accuracy loss due to the fixed-point operation, the precision loss is bearable. Besides, we compare the results of intermedia layers to find that the results generated by manually coded implementation and our design tool are identical.

6 Conclusion

This paper presents a design tool, CNNBUILDER, to simplify the design flow of CNN-based accelerators for machine learning and extend the versatility of the CNN-based accelerators. Our approach makes it easy for software developers to compose CNN models and implement their applications. Our approach adopts a unified data structure to store the information of different CNN models and then map them into FPGA. With our design tool, application developers without FPGA design experience can easily implement their design on the energy-efficient FPGA platform containing both FPGA and CPU. Meanwhile, we show that the accuracy of designs from our approach is guaranteed with only minor overhead in run time and power consumption. Our design tool improves the productivity of CNN based accelerator implementation by significantly reducing the time required to modify designs manually for new models.

Acknowledgment. This work is partially supported by National Key Research & Development Program of China (2017YFA0206104), Shanghai Municipal Science and Technology Commission and Commercial Aircraft Corporation of China, Ltd. (COMAC) (175111105000), Shanghai Municipal Science and Technology Commission (18511111302, 18511103502), Key Foreign Cooperation Projects of Bureau of International Co-operation Chinese Academy of Sciences (184131KYSB20160018) and Shenzhen Corerain Technologies Co. Ltd.

References

1. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
2. Zhao, R., Niu, X., Wu, Y., Luk, W., Liu, Q.: Optimizing CNN-based object detection algorithms on embedded FPGA platforms. In: Wong, S., Beck, A.C., Bertels, K., Carro, L. (eds.) ARC 2017. LNCS, vol. 10216, pp. 255–267. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56258-2_22
3. Ronneberger, O., Fischer, P., Brox, T.: U-Net: convolutional networks for biomedical image segmentation. In: Navab, N., Hornegger, J., Wells, W.M., Frangi, Alejandro F. (eds.) MICCAI 2015. LNCS, vol. 9351, pp. 234–241. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24574-4_28
4. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR 2014, pp. 3431–3440 (2015)
5. Liu, W., et al.: SSD: single shot multibox detector. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9905, pp. 21–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46448-0_2
6. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556v6 (2014)
7. Abdelouahab, K., et al.: Accelerating CNN inference on FPGAs: a survey (2018)
8. Lacey, G., Taylor, G.W., Areibi, S.: Deep learning on FPGAs: past, present, and future. arXiv e-print 2 (2016)
9. Sharma, H., et al.: From high-level deep neural models to FPGAs. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, pp. 1–12 (2016)
10. Wang, Y., et al.: DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In: Design Automation Conference, pp. 1–16. IEEE (2016)
11. FDDB: A Benchmark for Face Detection in Unconstrained Settings. Technical Report UM-CS-2010-009, Deptartment of Computer Science, University of Massachusetts, Amherst (2010)