A COMPILER DESIGN FOR A PROGRAMMABLE CNN ACCELERATOR

Jiadong Qian, Zhongcheng Huang, and Lingli Wang[*]
School of Microelectronics, Fudan University, Shanghai 201203, China
*Corresponding Author's Email: llwang@fudan.edu.cn

ABSTRACT

Convolutional Neural Networks (CNNs) are widely used in many AI applications, such as image classification, target detection, and target tracking. Due to the increase of CNN computational complexity, hardware acceleration is necessary for inference. Programmable accelerators are promising because of their support for different CNN models. To program an existing programmable accelerator, dedicated instructions need to be generated. In this paper, a compiler is designed to generate the instructions. The compiler explores the best partition of CNN models, schedules the sequence of computing, and generates the instructions automatically. With the proposed compiler, the instruction-driven CNN accelerator achieves the throughput varied from 114 FPS (ResNet152) to 1130 FPS (AlexNet).

INTRODUCTION

High performance has been achieved in multiple AI applications based on CNNs, such as computer vision, robotics, and natural language processing. However, the accuracy and capability of CNNs come at the expense of computational complexity. Therefore, some researches focus on the deployment of CNNs on hardware platforms such as GPUs, FPGAs or CGRAs[1] for acceleration. Corresponding configuration for programmable accelerators on these platforms must be provided according to different CNN models. In [1], configuration for an accelerator based on CGRA is generated manually, which costs much time and effort. Automatic compilers, such as TVM[1], DLA[3], fpgaConvNet[4], are also compared in [5]. TVM is a powerful compiler which can deploy CNNs to different hardware platforms, such as CPUs, GPUs and VTA[6], a programmable accelerator on FPGAs. However, VTA on the FPGA Ultra-96 only outperforms the Cortex-A53 by 3.8x on ResNet50. DLA presents a compiler and FPGA overlay for Neural Networks acceleration. It proposes a very long instruction word (VLIW) but introduces some overhead.

A compiler which can generate different hardware configuration sequences according to different network structures is proposed in this paper with the following contributions:

- The general accelerator architecture is abstracted in the analysis of the existing CNN accelerators [1] [6].
- The method of partitioning and scheduling is proposed according to the methodology of CNN calculation.
- The functional of the compiler is verified and the performance of the accelerator is evaluated on Xilinx VCU118.

THE HARDWARE ARCTHITECTURE

The high-level overview of the programmable accelerator architecture is proposed as shown in *Figure 1*. The accelerator is composed of four modules:

- The *instruction schedule* module fetches the instructions from the DRAM, decodes and passes them to the other three modules according to the type of the instructions.
- The *load* module loads input feature maps, weights, and biases from the DRAM.
- The *compute* module can perform various calculations such as *Convolution*, *Pooling*, *Activation*, etc.
- The *store* module stores results produced by the compute module back to the intermediate buffer or the DRAM.

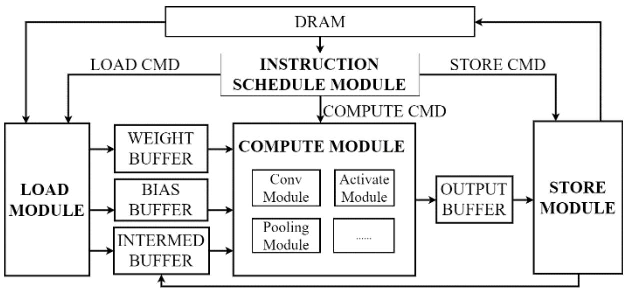


Figure 1: The hardware architecture of a programmable CNN accelerator

THE SOFTWARE FLOW

According to the function, the compiler can be divided into the following two parts:

- Partitioning: The partitioner divides layers of CNN into multiple computing blocks.
- Scheduling: The scheduler schedules the block sequence and generates specific instructions.

The compiling flow is shown in *Figure 2*.

Partitioning

Because of the limited hardware computing and storage resources, the hardware modules have to be reused during the inference of CNN. Data needs to be loaded, computed and stored multiple times. The input feature map of each layer is divided into small Data Blocks(DBs) of the

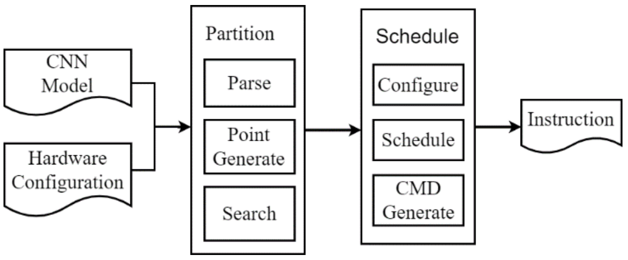


Figure 2: The flow of the CNN compiler

same size, which is called partitioning. As shown in *Figure 3*, the notations are defined as the height of input blocks(*bih*), the channel of input blocks(*bic*), the height of output blocks(*boh*), the channel of output blocks(*boc*).

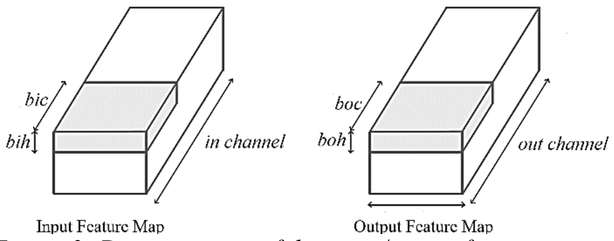


Figure 3: Data partition of the input/output feature map

Partitioning can be described as an optimization problem: with a set of variables, find the lowest cost solution under a series of constraints. The variables, which are represented by a quad-tuple <*mode*, *bic, boc, boh*>, include the degree of parallelism and the size of the DBs. The constraint in the partitioning is that the size of a DB should be smaller than the size of the on-chip buffers. The cost to be minimized is the total time to calculate a convolution layer, while the cost of computing a single DB is defined as:

$$Cost_{block} = Cost_{load} + Cost_{compute} + Cost_{store} \quad (1)$$

the cost of load and store could be defined as:

$$Cost_{io} = \frac{DB\ Size}{BW} \quad (2)$$

where *DB Size* is the total amount of data and BW is the IO bandwidth. The cost of compute is defined as:

$$Cost_{compute} = \frac{T_c}{f} \quad (3)$$

where T_c is the clock cycle number of computing a DB and f is the clock rate of the computing core in the accelerator.

Finally, the partitioning flow is summarized in *Figure 4*. A quad-tuple set, which represents the search space, is generated by an iterator in the compiler. Then the tuple set, the constraint function and the cost function are passed to the searcher. The searcher traverses the tuple set to find the lowest cost solution under the constraints.

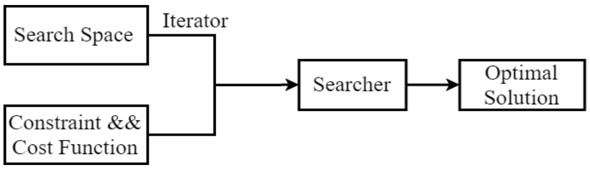


Figure 4: Partitioning flow

Scheduling

The goal of scheduling is to increase throughput. The purposed runtime distribution of the instructions is shown in *Figure 5*. Three types of instructions of the accelerator are described as follows:

- *Load*: fetches the data (feature maps, biases, weights) from DRAM and writes it to on-chip buffers.
- *Store*: writes the data in the output buffer to DDR.
- *Compute*: performs computing functions.

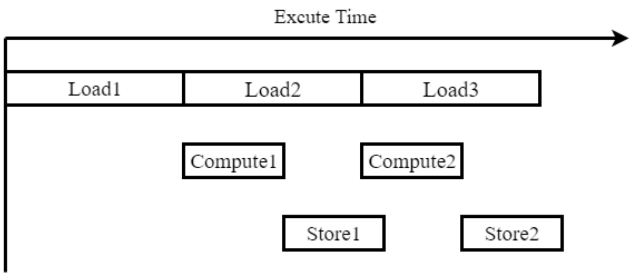


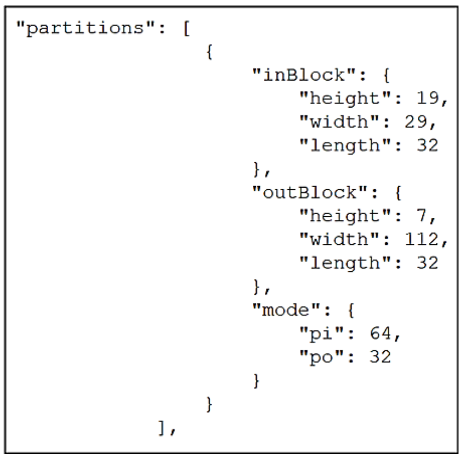
Figure 5: Purposed runtime distribution of the instructions

The scheduling is implemented with two steps. The first step is the Computing Blocks (CBs) generation: according to the partition results, the scheduler generates a series of CBs which includes some information needed for computing. The second step is to take the CBs as input and perform the scheduling. This step mainly considers the logic of the runtime, and launches the instructions to the accelerator as soon as possible. Hence, the hardware performance can be maximized.

EXPERIMENTAL RESULT

This paper implements a compiler with an existing programmable CNN accelerator which matches all the descriptions of the previous hardware architecture. For example, the partial partitioning result of ResNet50 is shown in *Figure 6*.

In order to verify the compiler, this paper uses the instructions generated by the compiler to emulate on the CNN accelerator. All of the simulation results are the same as the results inferred by Caffe. To evaluate the performance of scheduling, we test the idle ratio of Resnet50,



```
"partitions": [
                {
                    "inBlock": {
                        "height": 19,
                        "width": 29,
                        "length": 32
                    },
                    "outBlock": {
                        "height": 7,
                        "width": 112,
                        "length": 32
                    },
                    "mode": {
                        "pi": 64,
                        "po": 32
                    }
                }
            ],
```

Figure 6: Partitioning result of one layer of ResNet50

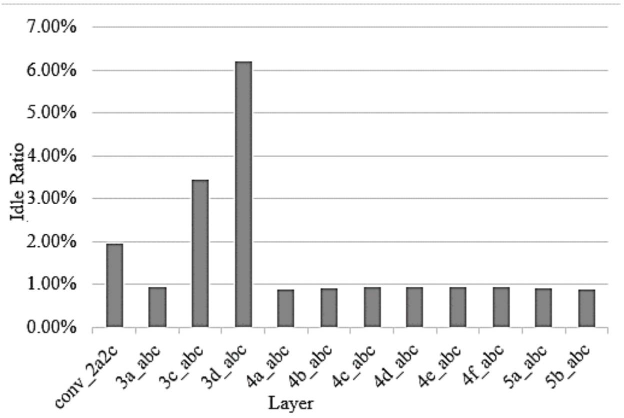


Figure 7: Idle Cycle Ratio of ResNet50

the partial result is shown in *Figure 7*, where the idle ratio is defined as follow:

$$Idle\ ratio = \frac{total\ cycles - effective\ cycles}{total\ cycles} \quad (4)$$

Layer conv_2a2c in *Figure 7* includes conv1, pool1, res2a_abc, res2b_abc, res2c_abc.

The programmable accelerator is implemented on Xilinx VCU118 at the clock rate of 400 MHz. The data is quantized to 8-bit fixed-point representation. The performance of the accelerator and the evaluation results on the ImageNet dataset are shown in Table I. We also compare our compiler with other compilers for FPGA-based CNN accelerator. The result is shown in Table II, with DNNVM for Deephi DPU on ZCU102 at 330 MHz, xfDNN of Xilinx on VU9P at 450 MHz, and fpgaConvNet on ZC706 at 125 MHz.

CONCLUSION

In this paper, the partitioning and scheduling methods of CNN calculation are proposed, and the corresponding compiler, which can automatically deploy CNNs on the specific programmable accelerator, is realized. Using this

TABLE I. PERFORMANCE OF DIFFERENT CNN MODELS

Model	Top-1 (%)	Top-5 (%)	FPS
ResNet50	72.6	90.9	222
ResNet101	73.7	91.6	151
ResNet152	74.7	92.3	114
AlexNet	58.7	81.2	1130

TABLE II. COMPARISON OF DIFFERENT COMPILERS

Model	DNNVM [5]	xfDNN [5]	fpgaConvNet [5]	This paper
ResNet50	74	80.5	N/A	222
ResNet152	27.5	28.7	6.5	114

(Unit: FPS)

compiler, the implementation process of CNN hardware acceleration can be automatic, and the accelerator can compute efficiently after the scheduling. A potential direction for further research is to develop the compiler and accelerator to support more different models.

ACKONWLEDGE

This work is supported by the National Natural Science Foundation of China under grant 61971143.

REFERENCES

[1] X. Fan et al., "Stream Processing Dual-Track CGRA for Object Inference," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018, pp. 1098-1111.

[2] T. Chen et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, Berkeley, 2018, pp. 579-594.

[3] M. S. Abdelfattah et al., "DLA: Compiler and FPGA Overlay for Neural Network Inference Ac-celeration," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL),* Dublin, 2018, pp. 411-4117.

[4] S. I. Venieris and C. Bouganis, "fpgaConvNet: Mapping Regular and Irregular Convolutional Neu-ral Networks on FPGAs," in *IEEE Transactions on Neural Networks and Learning Systems*, 2019, pp. 326-342,

[5] Y. Xing et al., "An In-depth Comparison of Compil-ers for Deep Neural Networks on Hardware," in *2019 IEEE International Conference on Embedded Soft-ware and Systems*, Las Vegas, 2019, pp. 1-8.

[6] Moreau T. et al., "A Hardware-Software Blueprint for Flexible Deep Learning Specialization," in *arXiv*, 2018, 1807.04188