Efficient compiler code generation for Deep Learning Snowflake co-processor

Andre Xian Ming Chang
FWDNXT
achang@fwdnxt.com

Aliasger Zaidy
FWDNXT
azaidy@fwdnxt.com

Eugenio Culurciello
FWDNXT
euge@fwdnxt.com

ABSTRACT

Deep Neural Networks (DNNs) are widely used in various applications including image classification, semantic segmentation and natural language processing. Various DNN models were developed to achieve high accuracy on different tasks. Efficiently mapping the workflow of those models onto custom accelerators requires a programmable hardware and a custom compiler. In this work, we use Snowflake, which is a programmable DNN targeted accelerator. We also present a compiler that correctly generated code for Snowflake. Our system were evaluated on various convolution layers present in AlexNet, ResNet and LightCNN. Snowflake with 256 processing units was implemented on Xilinx FPGA, and it achieved 70 frames/s for AlexNet without linear layers.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Hardware** → *Hardware accelerators*;

KEYWORDS

Deep learning, neural networks, co-processor, compiler

1 INTRODUCTION

DNNs are composed of various stacked layers of multi dimensional tensor operations, making them computationally intensive and power hungry. Supervised learning methods are two steps process: training and inference. Training is done once in a while when the user wants to modify the DNN model to adapt to a new data pattern. Currently, training requires several iterations and high volume of data. Server based systems thrive on this type of workload, due to its superior raw performance and storage capacity. On the other hand, inference phase requires in real time execution and it doesn't require high data precision [10]. Inference hardware accelerators have proliferated, due to the demand of real time DNN inference on embedded systems. Custom hardware achieves superior performance per power consumed making them well-suited for deploying DNNs. Several hardware accelerators that use ASIC or FPGA were developed [1, 2, 6, 13].

This paper contributes to the field by introducing software support for Snowflake [4], which is a programmable low-power accelerator for DNNs. Snowflake architecture was designed to provide high computational occupancy, as long as it gets an optimal sequence of instructions. The purpose of the compiler is to generate correct and efficient code for Snowflake.

Snowflake with 256 processing units running at 200 MHz was implemented on Xilinx's ZC706 development board [12]. And we were able to execute AlexNet, ResNet and LightCNN [5, 7, 9, 11].

2 OVERVIEW

This section presents an overview of convolution layers and Snowflake's architecture.

2.1 Convolution

Convolutional Neural Networks (CNNs) are the main type of DNN for tackling image processing tasks. Convolution (CONV) is the core computation of CNNs workloads. It has two operands: 3D input (IN) and 4D weight (W), and it produces a 3D output (OUT). Each output plane (outP) is the result of convolving IN with a 3D weight from W. Each 3D weight (K) is associated with a bias value. In other words, all values in an output plane has the same bias value. A section of IN and a respective section of K are multiplied and accumulated to produce a value of OUT. This section dimensions are: kernel height (kH), width (kW) and input plane (inP). Most CONV layers have $inP > kH \times kW$. Thus, arranging IN and W along z-axis first exhibits better data locality. This section is then strided to produce the next output value. CONV also defines padding on the corners of the IN. A good visualization of CONV is shown in [3].

2.2 Snowflake

Snowflake was presented in [4]. For readers' convenience, an overview of Snowflake is presented in this section.

Snowflake's primary compute engine is a 16 bit multiply and accumulate units (MACs). The data precision of choice

EMC[2], Mar 2018, Williamsburg, VA, USA

978-1-5386-7367-6/18/$31.00 ©2018 IEEE
DOI 10.1109/EMC2.2018.00013

CPS Conference Publishing Services

is fixed point Q8.8. 16 MACs form a vector MAC (vMAC). A compute unit (CU) is composed of 4 vMACs and a vector max-pool unit (vMAX). 4 CUs are grouped to form a compute cluster (CC). Every vMAC in a CU shares has one shared maps buffer (MBuf) and each vMAC has a private kernel buffer (WBuf). Snowflake instructions are stored in the instruction buffer (I$). Snowflake's control unit is a pipeline that executes those custom instructions. The control unit provides 32× 32 bit scalar registers. Each CC has its own control unit and I$. The implemented Snowflake has 512 KB of WBuf and 256 KB of MBuf and 4 KB of I$. Snowflake with 256 processing units running at 187 MHz was implemented on AC510, which has HMC memory and Xilinx's KU060 [8].

Snowflake's custom ISA has 13 different 32 bit instructions. The details on each instruction is presented in [4]. They mainly implement 4 functionalities: data movement, compute, flow control and memory access. The most relevant instructions are: MACV, MAX, VMOV, LD, TMOV and scalar instructions.

MACV multiplies and accumulates a sequence of data from MBuf and WBuf. This sequence is called a trace. In CONV, a trace is usually $kW \times inP$. MAX compares a block of data from MBuf with another block of data. MACV and MAX store results back to MBuf. VMOV loads the MACs with a initial value. It sets the initial value of the accumulation. LD sends data from main memory to Mbuf, WBuf or I$. There are two LD modes: broadcast or individual buffer loads. This allows the processing choice of same weights and different maps, or same maps with different weights. TMOV sends data from MBuf to main memory. There are scalar instructions MOV, ADD, MUL and Branches. Together they implement necessary bookkeeping functionalities: loops, if else conditions, address increment/decrement and others.

MACV has 2 important modes of operation that were presented in [4], but it is worth recapitulating here, since each MACV mode presents different data access and compute patterns.

In cooperative mode (COOP), all MACs in one vMAC work together to produce 1 output value. Each MAC processes a different inP of K, and the results of all 16 MACs are added together. Each vMAC produces an different outP, so each WBuf contains a different K. Using 4 CUs, 16 values are produced, each belongs to a different outP. COOP mode needs a contiguous processing length (trace) bigger than and multiple of 16. Otherwise, independent mode (INDP) comes to action. In INDP mode, all MACs in one vMAC work independently on different K to produce 16 different output values. Thus, 4 vMACs produce 64 output values for different outP. Each CU in INDP mode has a different row section of IN. Independent mode is useful when a CONV has small inP. For example, most models initial layers need to extract features from an input image of inP = 3 (RGB).

3 CODE GENERATION

The compiler generates a stream of Snowflake's instructions given a DNN model defined in *ONNX*[1]. The workflow starts at reading a ONNX model file using *Thnets*[2]. This creates a list of layers to be ran on Snowflake. Each element in the list has input size, output size, kernel size, stride, padding and other parameters.

After parsing a model, code generation distributes the data and workload to CUs, and creates Snowflake instructions accordingly. The goal is maximum utilization of all CUs and their MAC units. In COOP mode, different Ks are loaded to all 16 vMACs and IN is broad cast to all MBuf. In this case, all CUs are used to create different values in outP. Most of the DNN layers have their inP and outP as a multiple of 16. In INDP mode, the computation flow is different. For example, consider first layer of AlexnetOWT [7], which is a CONV with height 224, width 224 and inP 3. Its output height 55, width 55, outP 64, kW 11, kH 11 and stride is 4 and padding is 2. This CONV is done in INDP mode. Each vMAC produces 16 values corresponding to a different outP, thus each WBuf have 16 different K. A CU creates all 64 outP, so each CU is responsible for a different section of the 55×55 output area.

In both COOP and INDP mode, maps and weights are divided into tiles that are sent from memory to MBuf and WBuf. Each tile involves up to 3 nested loops to produce an output section: a loop to stride the computation along y-axis (Y-Loop), a loop to stride x-axis (X-Loop), and a loop that covers all kH and kW to produce a single output value (T-Loop). Another two loops to go through all maps and weight tiles are needed to produce entire output of the layer. Arranging these loops present different levels of data reuse. In this work we discuss the arrangement benefits of the last two loops.

Loop rearrangement is a method that reduces total amount of data movement by exploiting data reuse, which leads to memory bandwidth savings. Maps and kernels need to be partitioned and processed in Mbuf and WBuf sized tiles. All kernel tiles need to be brought to WBuf for each map tile (reuse K), leading to repeated kernel loads when the next map tile is loaded. Alternatively, all map tiles need to be brought to MBuf for each kernel tile (reuse M). The total amount of data moved is different depending on kernel/map load repetition for a particular CONV layer.

Figure 1 shows an example of the generated code for a CONV layer. Part a) presents a visualization of the code in part b). The compiler creates the initial loads to MBuf and WBuf with data for the first set of computations. The load for the next tile is pre fetched, so that compute and load are overlapped in Snowflake. Then it creates a kernel loop

[1]https://github.com/onnx/onnx

[2]https://github.com/mvitez/thnets

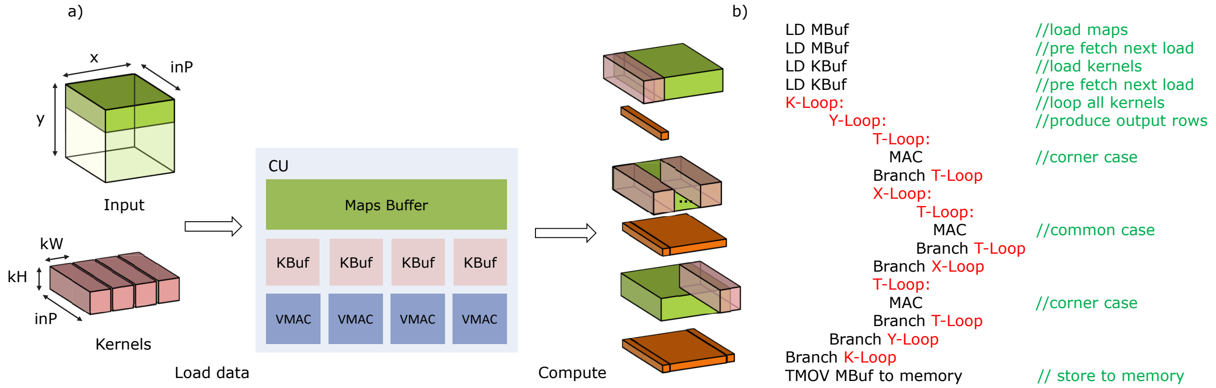

Figure 1: Example of computation of CONV in Snowflake. a) Part of the input and part of kernel are loaded to Maps Buffer and KBuf. Then each kernel section is multiplied and accumulated through a section of the input to create output. Corner cases have a different section size that produces a output value. b) Shows an example of the generated instructions.

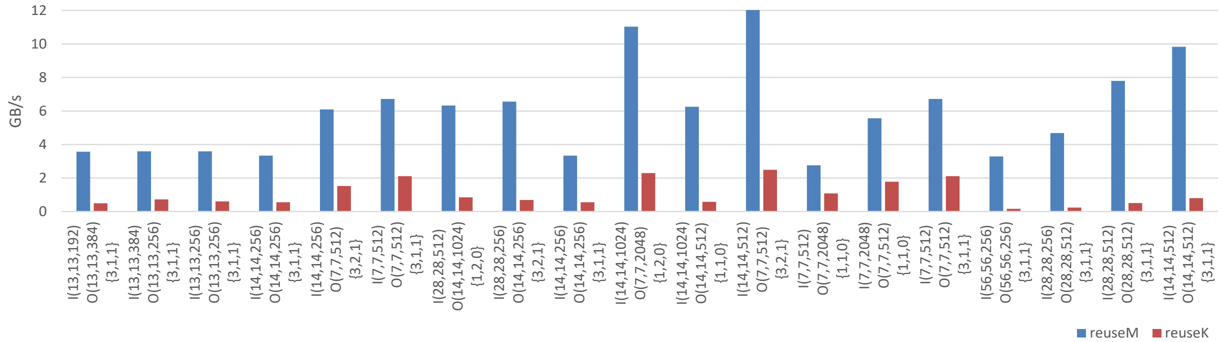

Figure 2: Required memory bandwidth by reuseM or reuseK mode for various CONV examples.

(K-loop) to go through all the kernels in KBuf producing multiple output channels. In this example, there are 2 corner cases. Each corner case have a different trace area ($kW \times inP$). T-Loop multiplies and accumulates all traces to produce a output value. The common case traces are ran inside of X-Loop, which strides the traces along the x-axis. The last corner case has its own T-Loop to create another output value. This computation row is inside of a Y-Loop, which strides those instructions along the y-axis. Finally, a TMOV instruction stores the results back to memory.

After the code generation phase, the last software task is to run the code on Snowflake. The Configuration registers are used to send an initial load instruction to populate Snowflake's I$ with the first set of instructions. Then the software polls an output counter register to check if Snowflake has finished its computation or not. A python interface was created so that user can import Snowflake support into their applications.

4 RESULTS

In this work, Snowflake has 512 KB of WBuf and 256 KB of MBuf and 4 KB of I$, with 1 CC running at 200 MHz, implemented on Xilinx's ZC706 development board. The results for generated instruction for different DNN models are shown in table 1. These numbers were measured using a Snowflake with 4 CUs running at 200 MHz. The execution time does not account for linear layers. This was implemented on Xilinx's ZC706 development board, which has a Zynq-SoC XC7Z045 FPGA [12]. $224 \times 224 \times 3$ images was used as input, except for LightCNN9 [11], which uses $128 \times 128 \times 1$ images for face identification.

The compiler took 25 s to generate Snowflake code for VGG16. And it took 15 s for ResNet50. The compiler was executed in the Zynq XC7Z045 embedded ARM Cortex-A9 at 666 MHz.

Table 1: DNN inference time on Snowflake.

Model	Time [ms]	Frames/s
AlexNet	14.0	71
ResNet18	58.3	17
ResNet34	106.0	9
ResNet50	149.8	6
LightCNN9	28.9	35

4.1 Instruction analysis

The instructions breakdown of each network is shown in figure 3. As expected the number of bookkeeping instructions (ADD and MOV) is larger than the number of vector instructions (LD, MAC, MAX, VMOV and TMOV). ResNet models have more VMOV instructions, since it has residual add layers that uses VMOV to do the vector add. The number of MAX is almost negligible compared to number of MAC instructions, because CONV layers are the majority in DNNs. Table 2 shows that majority of CONV layers are executed in COOP mode, thus most of the layers have inP as a multiple of 16. LD length is in 64 B granularity, so LD length of 1 transfers 64 B of data. The max of LD length is expected to be the size of one buffer of the MBuf $32KB/64B = 512$. The MAC trace length is represented with 12 bit so their max is 4096. CONV layers in the benchmark models don't have $kW \times inP$ larger than 2048.

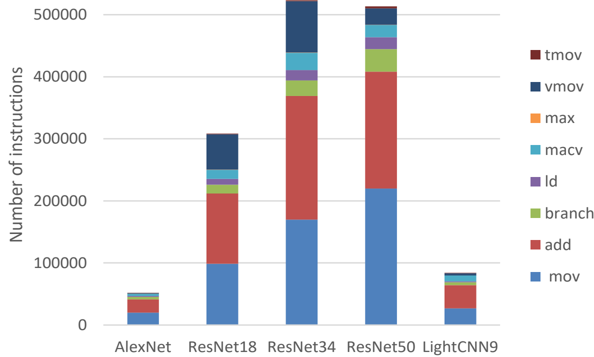

Figure 3: Number of instructions break down per DNN model.

The compiler achieves performance comparable to handcrafted instructions as shown in table 3. The results in this table were measured with an early prototype Snowflake system with 1 CU system, running at 142 MHz. In the table, CONVs parameters are, respectively, input size, kernel size, input plane, output plane, stride and padding. *Auto* stands for compiler generated code and *hand* is handwritten code.

Table 2: Instruction parameters in different models. LD max is largest LD length. MAC max is largest trace length for a MAC

Model	INDP	COOP	LD max	MAC max
AlexNet	96	3652	468	1152
ResNet18	263	14336	336	1536
ResNet34	263	27758	336	1536
ResNet50	535	19079	448	2048
LightCNN9	764	8454	384	576

Auto-generated code has higher instruction count (437 more), but it achieves similar efficiency to hand optimized code. Efficiency is the ratio between expected time with measured run-time. Expected time is calculated with equation 1, where *MACunits* is total number of MAC units in Snowflake. Each MAC unit can do 2 Ops per cycle.

$$Expected_time = \frac{Ops}{2.MACunits.freq} \tag{1}$$

Table 3: Hand optimized code (hand) versus auto-generated instructions (auto) for some AlexNet layers.

Layer	Code	Time [ms]	Eff. [%]
13x13,3x3,192,384,1,1	Hand	11.11	99.5
	Auto	11.08	99.7
13x13,3x3,384,256,1,1	Hand	14.84	99.3
	Auto	14.77	99.8
13x13,3x3,256,256,1,1	Hand	9.89	99.3
	Auto	9.86	99.6

4.2 Loop rearrangement for bandwidth constraints

Unlike GPUs and ASIC designs, FPGA accelerators are limited mostly by their off-chip memory bandwidth. The required bandwidth for a layer is a ratio between total amount of data transfered by the expected execution time. The bandwidth requirements for some CONV layers are shown in figure 2. This shows that reuseK leads to lower memory bandwidth requirements for the CONVs present in most DNNs.

5 CONCLUSION

A programmable device provides flexibility to implement different workloads using the same hardware architecture. It is up to the compiler to generate code that exploits the full extent of the hardware capabilities. Identifying better code strategies are especially important for low power accelerators. This work presented Snowflake compiler that generates

code for various DNNs and analyses potential code improvements for running DNNs on Snowflake accelerator.

REFERENCES

[1] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2017. Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes. *arXiv preprint arXiv:1701.06420* (2017).

[2] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 367–379. https://doi.org/10.1145/3007787.3001177

[3] Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).

[4] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. 2017. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on.* IEEE, 1–4.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[6] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).

[7] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR* abs/1404.5997 (2014). http://arxiv.org/abs/1404.5997

[8] Micron 2017. *AC-510 UltraScale FPGA with Hybrid Memory Cube*. Micron. http://picocomputing.com/wp-content/uploads/2016/01/AC-510_Product_Brief.pdf

[9] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556

[10] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. 2015. Resiliency of deep neural networks under quantization. *arXiv preprint arXiv:1511.06488* (2015).

[11] Xiang Wu, Ran He, Zhenan Sun, and Tieniu Tan. 2015. A light CNN for deep face representation with noisy labels. *arXiv preprint arXiv:1511.02683* (2015).

[12] Xilinx 2015. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC*. Xilinx. http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf v1.5.

[13] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD '16)*. ACM, New York, NY, USA, Article 12, 8 pages. https://doi.org/10.1145/2966986.2967011