Deep Neural Networks Compiler for a Trace-Based Accelerator (Short WIP Paper)

Andre Xian Ming Chang
FWDNXT
West lafayette, USA
achang@fwdnxt.com

Aliasger Zaidy
FWDNXT
West lafayette, USA
azaidy@fwdnxt.com

Lukasz Burzawa
FWDNXT
West lafayette, USA
lburzawa@fwdnxt.com

Eugenio Culurciello
FWDNXT
West lafayette, USA
euge@fwdnxt.com

Abstract

Deep Neural Networks (DNNs) are the algorithm of choice for image processing applications. DNNs present highly parallel workloads that lead to the emergence of custom hardware accelerators. Deep Learning (DL) models specialized in different tasks require a programmable custom hardware and a compiler/mapper to efficiently translate different DNNs into an efficient dataflow in the accelerator. The goal of this paper is to present a compiler for running DNNs on Snowflake, which is a programmable hardware accelerator that targets DNNs. The compiler correctly generates instructions for various DL models: AlexNet, VGG, ResNet and LightCNN9. Snowflake, with a varying number of processing units, was implemented on FPGA to measure the compiler and Snowflake performance properties upon scaling up. The system achieves 70 frames/s and 4.5 GB/s of off-chip memory bandwidth for AlexNet without linear layers on Xilinx's Zynq-SoC XC7Z045 FPGA.

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

Keywords DNN, Compiler, accelerator

ACM Reference Format:
Andre Xian Ming Chang, Aliasger Zaidy, Lukasz Burzawa, and Eugenio Culurciello. 2018. Deep Neural Networks Compiler for a Trace-Based Accelerator (Short WIP Paper). In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3211332.3211333

LCTES'18, June 19–20, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5803-3/18/06.
https://doi.org/10.1145/3211332.3211333

1 Introduction

Deep Neural Networks (DNNs) are widely adopted in various areas, such as robotics, security and data analytics. DNNs are composed of various layers, each with a large parallel workload, making them well suited to hardware accelerators. Accelerators are a very attractive solution for deploying DNNs especially in the embedded world, where it demands real-time processing under a limited power budget. Several hardware accelerators that use ASIC or FPGA were developed [3, 6, 11, 15, 20].

This work uses Snowflake [8], which is a scalable, programmable and low-power accelerator. Snowflake's architecture was designed to achieve maximum computational occupancy; in other words, all its compute units should be active during the execution of different types of DNN workloads.

Snowflake implements a 32 bit RISC-like custom instruction set (ISA). Manually crafting assembly-like custom instructions can be cumbersome and error prone, especially when a model is composed of several layers. Even if one was willing to write Snowflake code, modifying thousands of lines would be required to further customize the hardware and software. The compiler should generate correct, optimal code. It also needs to parse various DNN models created from different high-level frameworks, so that users can use their favorite Deep Learning (DL) tool.

The goal of this paper is to present SnowC (Snowflake Compiler). It supports ONNX interchange format, allowing it to parse models from different frameworks. The compiler correctly generated Snowflake instructions for various DNN models: AlexNet[12], VGG [16], ResNet [9] and LightCNN, which is used for face identification applications [18]. SnowC also provides a python interface for users to create their demo applications, such as image categorization and face identification. For this work, Snowflake architecture was reproduced and implemented in two FPGA systems: AC510 [14] and ZC706 [19].

2 Snowflake

Snowflake was presented in [8]. Snowflake's main compute engine is a 16 bit multiply and accumulate unit (MAC). The data precision of choice of the MACs is fixed point Q8.8. A vector MAC (vMAC) is composed of 16 MACs, and 4 vMACs plus a vector max-pool unit (vMAX) are grouped into a compute unit (CU). Each vMAC has a private double buffered kernel buffer (WBuf) and every vMAC in a CU shares the input data through the maps buffer (Buf). A compute cluster (CC) is composed of 4 CUs, and multiple CCs can be implemented, each containing 256 MACs. Snowflake's 32 bit instructions are stored in the instruction buffer (I$). Each CC has a control unit, which is a pipeline that executes those instructions. The control unit has thirty two 32 bit registers to store scalar values. Snowflake's custom ISA has 13 different instructions, which implement four different functionalities: data movement, compute, flow control and memory access. The details of each instruction are presented in [8]. For this paper, the most relevant instructions are: MAC, MAX, VMOV, LD, TMOV and Branch. MAC multiplies and accumulates a contiguous sequence of data from Buf and WBuf. MAX compares two blocks of data from Buf. MAC and MAX send results back to Buf. VMOV pre-loads data to the compute unit to set the initial value for MAC. It is used to add the bias or implement the residual addition. LD sends data from external memory to Buf, WBuf or I$. TMOV sends data from Buf to external memory. Branch is used to create loops and if-else conditions.

3 Parsing

The first step towards generating code for a custom accelerator is to gather information about the model's architecture. There are various high level DL frameworks [2, 5, 7, 10] being used today. Each represents DNNs differently, and they exploit different optimizations for deploying it on CPU or GPU systems. An evaluation of those frameworks is presented in [4]. ONNX [1] is an intermediate exchange format that allows models from different frameworks to be converted into other formats. Adopting ONNX allows Snowflake users to deploy models that were trained on any DL framework supported by ONNX. Figure 1 shows the workflow to run a model on Snowflake.

Thnets [17] is used to read a model file exported from ONNX. A list of layer objects is created to represent the layer computation sequence in Snowflake. The objects contain information needed to generate code for Snowflake. For example, the vector add operation present in ResNet models is merged with a convolution layer. Non-linear operations, such as MFM in [18] and ReLU, are also merged with convolution layers. An example is shown in figure 2, parts 1 and 2. Currently, the layers that are supported are spatial convolution (CONV), transpose convolution (TCONV), fully connected layers (FC), max pooling (MAX), average pooling

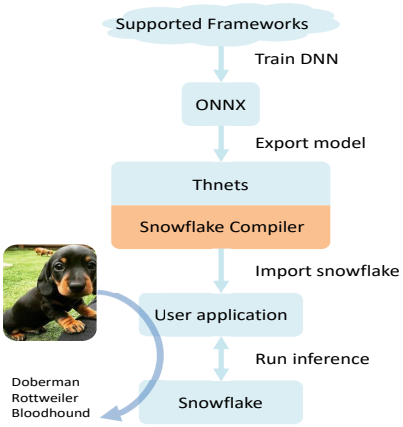

Figure 1. Work flow to run inference on Snowflake. DNN models are trained using a DL framework and imported to SnowC using thnets. User application calls SnowC functions to run on Snowflake.

(AVG), residual add (RESADD) and MFM. Support for other layers are under development.

4 Generating code

After the model parsing task completes, SnowC needs to partition the input and weights data and map the dataflow into the Snowflake architecture. Snowflake and other DNN accelerators [3, 13, 20], are composed of an on-chip memory buffer (Buf) to store data, a group of processing elements and a control core. This leads to 3 main operations: load, compute and store. In SnowC, a sequence of load, compute and store is grouped into a compute step. Each step consume part of the layer's input and produces part of the layer's output. The compiler creates a list of compute steps based on the layer parameters.

The limits imposed by Buf size and layer parameters are first calculated before creating the compute list. Based on these limits, load objects (LO) are created such that a balance between input data and output data coexists in the same Buf. LO sends data from external memory into Buf. Snowflake has a separate Buf for weights (WBuf) and the algorithm aggregates as many weights as possible that fit in WBuf. Double buffering is accounted for during LO creation, such that a compute step will pre-fetch data for the following compute step. After LO creation, compute objects (CO) are generated based on the data available in Buf and WBuf.

CO contains information about the vector compute operations necessary for producing a number of output values. This encompasses up to 3 loops: stride on y-axis, x-axis and accumulate. The accumulate loop issues multiple instructions that accumulate the results before producing an output pixel. This is because not all data needed to produce an output value is contiguous. CO will be translated into nested loops of vector instructions to perform multiply and

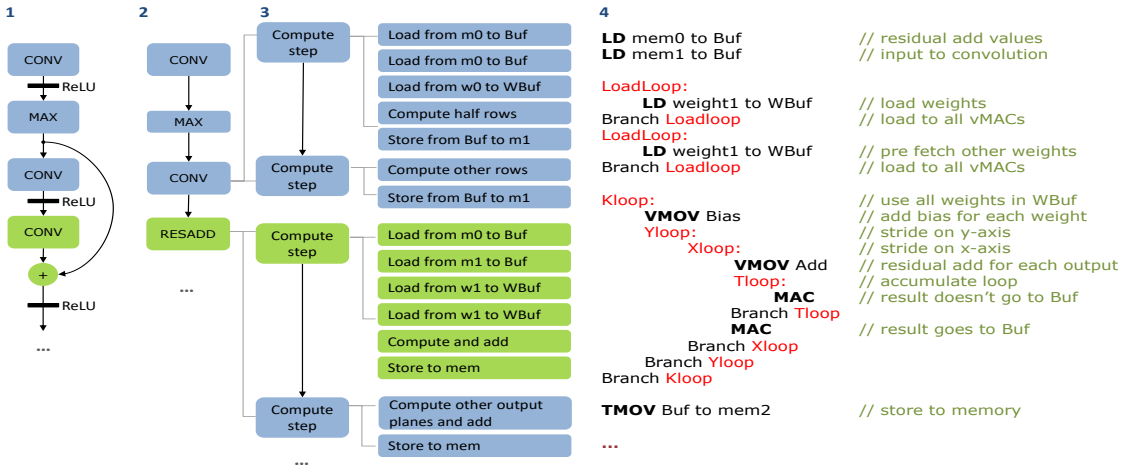

Figure 2. Example of instruction generation for part of ResNet model. 1- ResNet model layers. 2- list of SnowC layers. 3- Creation of compute steps that contains a list of LO, CO and SO. 4- Shows the main Snowflake instructions generated from a compute step.

accumulate or compare. The loop boundaries are a repeat variable and the data access address is incremented by an offset variable. CO also has an extension with variables for vector register load, which implements residual add and bias addition.

Lastly, store objects (SO) are created to return the output in Buf to external memory, so that the CPU can access the results. Compute step creation is shown in figure 2, part 3. In the example, assume input data to CONV is in m0, which is also the input to a residual add in a following RESADD layer. m1 is another memory location that has the output of the previous CONV and it is the input of the CONV part of the RESADD. w0 and w1 are other memory locations for weights. Precise address offsets were omitted.

Once a list of compute steps is created, a code generation phase converts each load, compute and store objects into a list of Snowflake instructions. This phase takes care of register assignment, branch delay slot filling, loop creation and unrolling. In a LO, if multiple loads have same address offset then a loop will be generated around the load. CO unrolls the accumulate loops if they are small enough. SO creates barriers to avoid data corruption. Load, vector compute and store instructions have variable latency and can be issued in parallel. In cases when there is a latency mismatch, a barrier is needed to ensure that data was written to Buf before starting the store back to external memory. Code generation for a compute step is shown in figure 2 part 4. The main Snowflake instructions are mentioned in 5.

Snowflake instruction cache is separated into 2 banks of 512 instructions. For each bank a load is needed to load instructions to other bank. The compiler will group all the instructions into groups of 512 ensuring that there isn't a branch from one instruction bank to another. Thus instructions in a loop cannot be separated into 2 banks.

Snowflake has multiple compute clusters, each with an independent control core, Buf and WBuf. There is one barrier instruction that synchronizes all clusters execution. For example, if a cluster finishes producing half of outputs in a CONV, it will wait at the barrier instruction for the other cluster to also react a barrier and complete the CONV layer before going to second layer. Another possibility is to send different inputs for all clusters in which barrier isn't needed.

It is possible to instantiate Snowflake on different FPGA cards with one host processor. In this case, a separate Snowflake object is created for each FPGA card. Different FPGAs can run different models or different inputs. Address reallocation table is used to modify instructions if the main memory addresses is changed. A functional simulator for Snowflake was created to debug the generated code.

5 Results

The Snowflake used in this work has 512 KB of WBuf and 256 KB of Buf and 4 KB of I$. Snowflake with 1 CC running at 200 MHz was implemented on Xilinx's ZC706 development board, which has a Zynq-SoC XC7Z045 FPGA [19]. Snowflake at 187 MHz was implemented on AC510, which has HMC memory and Xilinx's KU060 [14]. The performance achieved using varying number of MACs was measured for some DNN models as shown in figure 4. The execution time on Snowflake doesn't account for the linear layers. 1F refers to 1 FPGA system, 2F 2 FPGAs and 4F FPGAs. Input size of choice is $3x224x224$. Only for LightCNN9 input size is $1x128x128$. Snowflake is able to scale its performance across FPGA cards with increasing number of FPGAs.

The measured power consumption of the ZC706 was 10W, one AC510 FPGA was 24W, Tegra TX1 14W, Titan-X 154W. On Alexnet without linear layers, performance per power

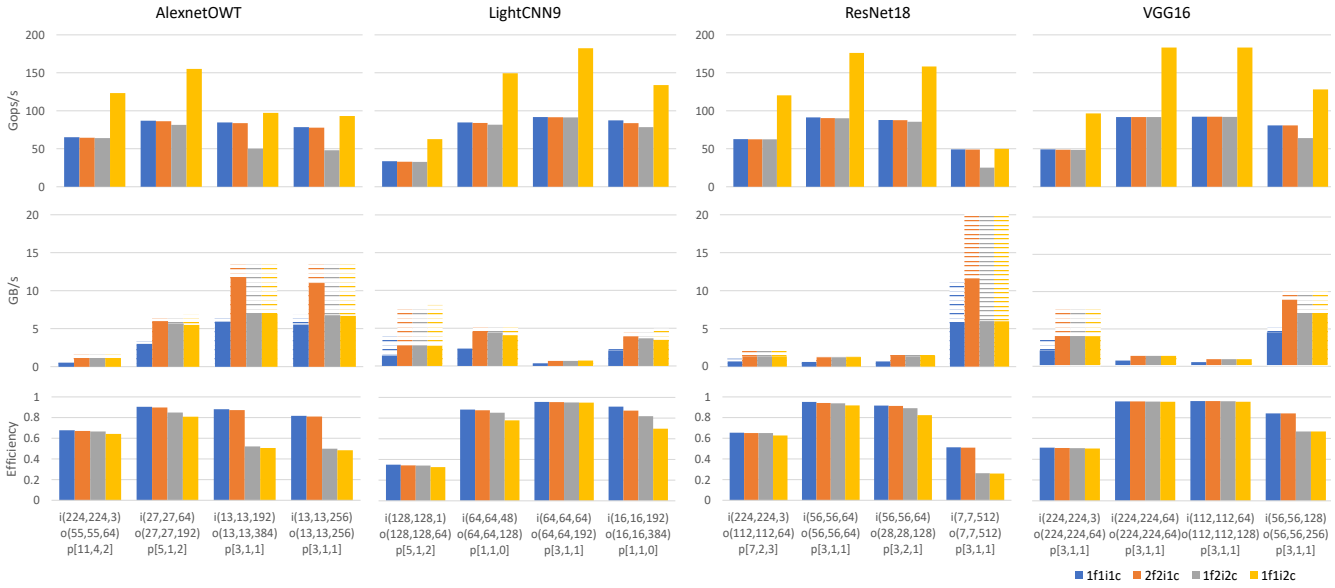

Figure 3. Plot of performance, bandwidth and efficiency for various convolution layers present in DNN models. 1f1i1c is 1 FPGA, 1 image and 1 CC. 2f2i1c is 2 FPGAs, 2 images and 1 CC each FPGA. 1f2i2c is 1 FPGA, 2 images and 2 CCs. 1f1i2c is 1 FPGA, 1 image and 2 CCs. In x-axis, i is input, o output and p parameters. i(224,224,3) is height 224, width 224 and 3 planes. p[11,4,2] is kernel height and width 11, stride 4 and pad 2.

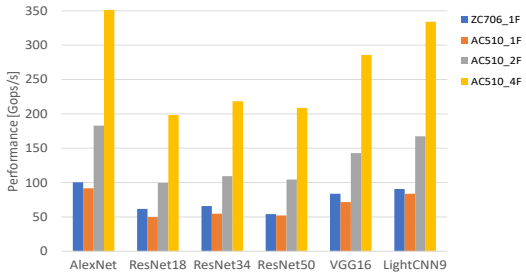

Figure 4. Execution time for running inference of different DNN models on different systems. 1F is one FPGA system, 2F two FPGAs and 4F four FPGAs.

consumed achieved in the ZC706 embedded platform was 4.5 Gops/W and in one AC510 FPGA was 3.8 Gops/W.

Figure 3 shows performance (top), bandwidth (middle) and efficiency (bottom) measured for some CONV layers in different DNNs. The x-axis shows the layer parameters: i is the input, o the output and p is the layer parameters. For example, i(224,224,3) is height 224, width 224 and 3 planes; o(224,224,3) is output height 55, width 55, and out plane 64; p[11,4,2] is kernel height and width 11, stride 4 and pad 2. The measurements were ran on 1 FPGA (1f) or 2 FPGAs (2f), using 1 input image (1i) or 2 images (2i) and using 1 CC (1c) or 2 CCs (2c). For example, in yellow 1f1i2c means that 1 image was distributed into 2 CCs within 1 FPGA. In gray, 1f2i2c 1 image was processed by each CC on 1 FPGA. Efficiency is calculated as ratio between measured execution time and expected execution time at peak performance. Memory bandwidth takes into account input, weights, output and instructions that are moved into/from Snowflake. The maximum bandwidth that we achieved on one FPGA is 7 GB/s. The bandwidth required for each layer is plotted in stripped bars.

The first layer of a network takes an image with input plane 3 or 1. Mapping these parameters to all CUs is not possible for all compute steps thus they achieve lower efficiency.

1f2i2c bandwidth requirement is higher because it has to send 2× of the input data using the bandwidth provided by 1 FPGA. In 2f2i1c, 2 FPGAs provide more bandwidth, thus it shows higher efficiency. 1f1i2c shows 2× performance boost as expected from using 2× more MACs on same number of operations. Improvements on SnowC algorithm to better exploit data reuse and reduce bandwidth requirements are under development.

6 Conclusion

This work presented a compiler that takes a model definition created with popular deep learning frameworks and produces code for a custom DNN accelerator. Together with Snowflake, this work contributes to the adoption of Snowflake in embedded or server based applications. Our future work involves improvements on code generation to achieve higher performance on Snowflake, and implementing support for more DNN models.

References

[1] 2017. Open Neural Network Exchange. (2017). https://github.com/onnx/onnx

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[3] 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).

[4] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. 2015. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435* (2015).

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[6] 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

[7] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[8] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. 2017 - in press. Snowflake: An Efficient Hardware Accelerator for Convolutional Neural Networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)*.

[9] 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

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[11] 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).

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

[13] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 393–405.

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

[15] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. 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)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/2847263.2847265

[16] 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

[17] Marko Vitez. 2017. Thnets. (2017). https://github.com/mvitez/thnets

[18] 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).

[19] 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.

[20] 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