A High Efficiency Accelerator for Deep Neural Networks

Aliasger Zaidy
FWDNXT Inc.
azaidy@fwdnxt.com

Andre Xian Ming Chang
FWDNXT Inc.
achang@fwdnxt.com

Vinayak Gokhale*
Google Inc.
drvgokhale@gmail.com

Eugenio Culurciello
FWDNXT Inc.
euge@fwdnxt.com

Abstract

Deep Neural Networks (DNNs) are the current state of the art for various tasks such as object detection, natural language processing and semantic segmentation. These networks are massively parallel, hierarchical models with each level of hierarchy performing millions of operations on a single input. The enormous amount of parallel computation makes these DNNs suitable for custom acceleration. Custom accelerators can provide real time inference of DNNs at low power thus enabling widespread embedded deployment. In this paper, we present Snowflake, a high efficiency, low power accelerator for DNNs. Snowflake was designed to achieve optimum occupancy at low bandwidths and it is agnostic to the network architecture. Snowflake was implemented on the Xilinx Zynq XC7Z045 APSoC and achieves a peak performance of 128 G-ops/s. Snowflake is able to maintain a throughput of 98 FPS on AlexNet while averaging 1.2 GB/s of memory bandwidth.

CCS Concepts • **Computer systems organization → Single instruction, multiple data**; • **Hardware → Emerging architectures**; *Hardware accelerators*;

Keywords Deep Neural Network Accelerator, Parallel Computing

1 Introduction

Deep Learning algorithms are being increasingly used for several tasks such as object localization [1–4], scene understanding [5, 6] and language translation [7]. Convolutional Neural Networks (CNN) are a type of deep learning algorithm that use a cascade of convolutional, hierarchical feature extractors to process an input signal and produce a result. These models perform billions of operations to produce a single output and consume a lot of power in the process. For a CNN, images are the most commonly used input. Recently, certain CNNs have claimed better classification accuracy than humans on specific tasks [8].

CNNs are massively parallel workloads that make them an excellent target for custom accelerators [9–12]. These accelerators enable energy efficient processing of CNNs compared to general purpose solutions. However, recent CNN accelerators fail to achieve high occupancy due to the variability in data access patterns across different CNN layers. Most accelerators manage to achieve high occupancy for a particular set of CNN layers but are unable to keep the compute elements occupied across all types of CNNs. In the following text, the terms computational efficiency or efficiency might be used interchangeably with occupancy.

This paper expands Snowflake[13], a high efficiency, low power accelerator for CNNs. Snowflake is extended to ensure scalability and improve utilization of memory bandwidth. Additional profiling was also performed on Snowflake. Snowflake is agnostic to the network architecture and was designed to achieve high occupancy of compute at all times. A secondary design objective of Snowflake was to achieve low memory bandwidth, thus, reducing power consumption. Several compression techniques [14–17] have been proposed for bandwidth bound CNN layers and can be used in conjunction with Snowflake to accelerate these layers. Our goal in designing Snowflake has been to efficiently map computations to the available resources. Hence, high computational efficiency is our primary design goal. Snowflake can achieve a computational efficiency of 91%-95% on various representative CNN workloads.

2 CNN Overview

CNNs are state of the art Deep Learning models with applications in domains like image search, autonomous driving, face identification, etc. These models are comprised of a cascade of hierarchical feature extractors called *layers*. The most common layer is one consisting of a series of convolution operations. Such a convolutional layer has two inputs: a 3 dimensional input maps volume and a 4 dimensional kernel volume. The convolutional layer also produces a 3 dimensional output maps volume. The input maps volume is the output of the previous layer or an image in case of the first layer of the network. Some convolutional layers also have a single bias value associated with each 3 dimensional part of

*Work done while at Purdue University, West Lafayette, Indiana

EMC², Mar 2018, Williamsburg, VA, USA

978-1-5386-7367-6/18/$31.00 ©2018 IEEE
DOI 10.1109/EMC2.2018.00010

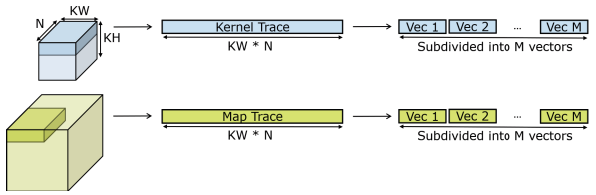

Figure 1. A trace is a contiguous region of memory necessary to produce an output. One or more traces may be required to produce a single output. Traces are subdivided into 256-bit vectors. Data is read from on-chip buffers at vector granularity.

the kernel volume. Some convolutional layers are also followed by a downsampling operator like spatial maxpooling or average pooling. Downsampling reduces the computational complexity of the network. [18] provides a detailed visualization of the operations involved in a CNN.

3 Microarchitecture

Snowflake was designed to achieve high occupancy of available compute resources. Multiply Accumulates account for most of the compute in DNNs. Keeping this in mind, Snowflake tries to achieve high occupancy of Multiply Accumulate (*MAC*) units. An important element for achieving high computational efficiency is data organization which is discussed in Section 3.1.

Snowflake is composed of three parts: (a) Control Core, (b) Compute Core, (c) Data Distribution Network. The Control Core is responsible for directing the vector operations performed by the Compute Core. It also enables data movement, synchronization and other bookkeeping operations. The Compute Core operates on the input data to enable low latency, high throughput processing of DNNs. The data distribution network is responsible for communicating data across the Compute Cores, fetching inputs from memory and writing results back to memory.

3.1 Data Organization

Data organization is a key factor enabling Snowflake to achieve the high occupancy necessary for efficient processing. Data is organized in Snowflake at the granularity of *traces*. A *trace*, shown in Figure 1, refers to a contiguous region of memory that is necessary to produce a single output. Computation and data movement in Snowflake occurs on *traces*. Depending on the size of the kernel, Snowflake requires one or more *traces* to produce a single output. As shown in the Figure 1, length of a trace is $kernelwidth(kW) \times depth$. For a kernel of size $3 \times 3 \times 384$, trace length is $3 \times 384 = 1152$ and 3 such traces are required to produce a single output. The Control Core is able to perform bookkeeping operation, such as address manipulation, while the compute resources are busy operating on a *trace*.

Traces are further sub divided into vectors. A vector is the granularity at which data is read from/written to on-chip buffers. Compute resources operate on a vector per cycle. Hence, a *trace* keeps compute resources occupied for several cycles. The size of a vector depends on the number of compute resources working on a trace in parallel i.e. the number of multiply accumulate units (*MACs*) present in a vector multiply accumulate unit (*vMAC*).

For the purpose of this publication, 16 *MACs* are used per *vMAC* and hence, a 256-bit vector size was used. Snowflake uses 16-bit fixed point format resulting in 16 words per vector. Loss of accuracy at 16-bit fixed point is negligible compared to 32-bit floating point for DNN Inference [19]. Snowflake performs computations on *traces* in two different modes: (a) Cooperative Mode, and (b) Independent Mode, which are explained further in Section 3.3. The use of *traces* does not require input data to be reorganized. Images can be directly fetched in RGB format. The intermediate results produced while processing the network are stored in the appropriate format by Snowflake.

3.2 Control Core

The control core is responsible for orchestrating the compute, data movement, synchronization and other auxiliary operations. Snowflake's control core is a 5 stage RISC like pipeline. The pipeline has a 4 kB instruction cache and provides 32x 32-bit scalar registers for metadata/address manipulation.

Instruction Set

Snowflake uses 32-bit wide instructions. Snowflake's ISA consists of two types of instructions: scalar instructions and vector instructions. Scalar instructions are used to keep track of addresses, for implementing branching primitives, and synchronization. Vector instructions include data movement instruction and compute instructions.

Snowflake's instruction comprises of the following fields:

1. *4-bit opcode* - to identify the type of operation being performed
2. *1-bit mode* - to distinguish between two instructions with the same opcode but slightly different operation (e.g. the move instruction and the move immediate instruction)
3. *an optional 5-bit destination register address* - to store the result of the operation
4. *one or more 5-bit source register addresses* - to hold the operands or a pointer to the operands
5. *an immediate field* - of varying length depending on the presence of other fields

The scalar instructions include data move instructions, add instructions, multiply instructions and branch instructions. Move Immediate *MOVI* and Move *MOV* are the data move instructions. The former is used to write a 22-bit immediate value into a scalar register while the latter is used to move

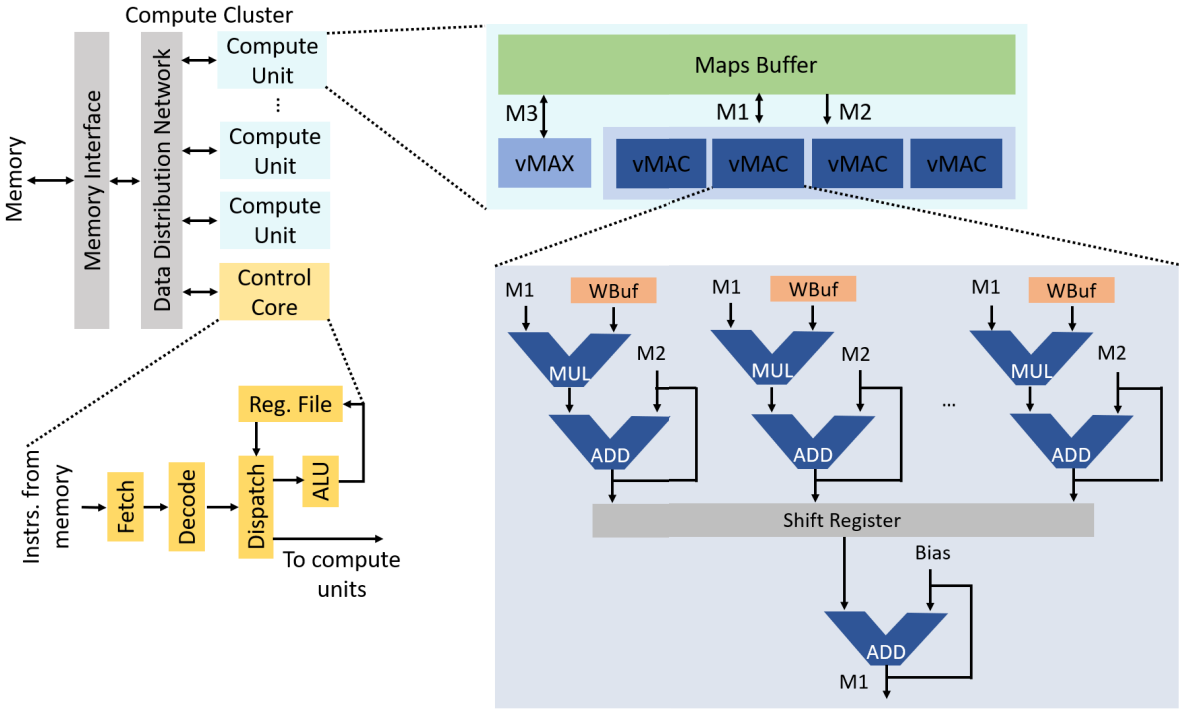

Figure 2. An overview of Snowflake. The three main components of Snowflake are: a) Control Core, b) Compute Core and c) Data Distribution Network. The Control Core is a 5 stage pipeline. The Compute Core consist of 4 Compute Units which contain 4 vector multiply accumulate units each. Each Compute Unit has a maps buffer shared between the *vMACs*. Each *vMAC* has its own weights buffer.

data from one scalar register to another. *MOV* allows data from the source register to be shifted before it is written into the destination register.

Similar to the move instruction, there are two types of add and multiply instructions:*ADDI*/*MULI* and *ADD*/*MUL*. The former adds/multiplies a scalar register with an immediate value, while the *ADD*/*MUL* instruction adds/multiplies two scalar registers.

Loops and binary conditional execution are commonplace in DNN code. Snowflake supports such control flow statements via conditional branch instructions. Snowflake provides three branch instructions: (a) Branch Less than Equal to (*BLE*), (b) Branch Greater than Equal to (*BGE*), and (c) Branch Equal to (*BEQ*). The branch instruction compares the contents of two source registers and takes action based on the successful evaluation of the expected condition. The target address can be +/- 1024 instructions from the branch instruction.

The vector instructions are Multiply Accumulate (*MACC*), Maxpool (*MAX*), Vector Register Move (*VMOV*), Load (*LD*), Store (*ST*) and Trace Move (*TMOV*). All vector instructions are trace based and contain a 12-bit immediate value to provide the trace length. The *MACC* instruction performs the convolution/matrix-multiply operations. The source registers for the MACC instruction contain the start address of the operand traces in the on chip buffers. Non linearities like ReLU, sigmoid, tanh, etc. are performed as a part of the *MACC* instruction. The *MAX* instruction performs a vector compare/maxpool operation. The Vector Register Move operation is used to preload data into the accumulator of the compute. This is important for loading bias values and for the bypass branch in case of ResNet [4]. The *LD* and *ST* instructions are used to communicate data/results from/to memory. The trace move instruction is used to move data across on chip buffers of different compute units.

3.3 Compute Core

For DNNs, Multiple Accumulates form the majority of operations. Hence, the basic compute element used in Snowflake is a multiply accumulate (*MAC*) unit. A *MAC* unit consists of a pipelined multiplier and adder. The multiplier processes two 16-bit input operands and produces a 32-bit result which is then accumulated by the adder. Pipelining ensures that the *MAC* unit can be clocked at a high frequency. Snowflake groups 16 such *MAC* units into a vector multiply accumulate unit (*vMAC*). Each *vMAC* has a 16 kB on chip buffer to store the weights operand. Four *vMACs* are combined to

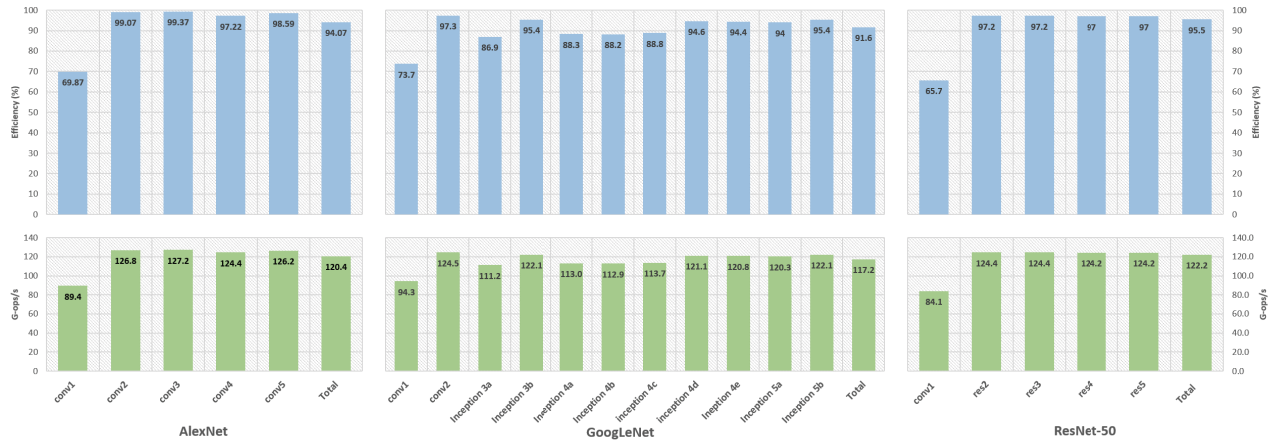

Figure 3. A single cluster Snowflake system was implemented on the Xilinx ZC706. The design contained 256 MAC units at 250 MHz. Snowflake achieves a computational efficiency of 91%-95% on various CNN benchmarks.

form a compute unit (*CU*) and four such *CUs* together constitute the compute core. The compute core is the workhorse of Snowflake. Each compute unit contains a 64 kB on chip buffer to store the maps operand. This maps buffer is shared by all *vMACs* within a *CU*. A *CU* also contains a vector compare unit (*vMAX*) to perform the maxpool. The *vMAX* unit fetches its operand from the maps buffer too. The results of *MACC* and *MAX* operations are stored back to the maps buffer.

Along with the compute, the *CU* also houses trace decoders to enable subdivision of traces into vectors. Trace decoders generate vector addresses that are fed to the on chip buffers which then deliver the data to the *vMACs* and *vMAX*. For a convolution, data is shared within and between the *vMACs* in two different ways. The data sharing patterns constitute the *Independent* and *Cooperative* Modes which are explained below:

Cooperative mode

In the cooperative (*COOP*) mode, the *MACs* within a *vMAC* work together on the same trace to produce a single output result. Each *vMAC* has a different kernel stored in its associated weights buffers. The maps vector is broadcast to all the *vMACs* within the *CU*. At the end of the computation, the *COOP* mode produces 16 partial results which are then reduced by a separate pipelined adder, called the gather adder, to produce the final output pixel. The *COOP* mode provides lower latency than the *Independent* mode and is suitable when the trace length is a multiple of number of *MACs* in a *vMAC* (16). A *CU*, in the *COOP* mode, produces 4 results at a time.

Independent mode

For some layers of a DNN, multiple of 16 trace lengths are not possible e.g. first layer of most CNNs. In such a case, Independent (*INDEP*) Mode is used. As the name suggests, the *MACs* within a *vMAC* work on independent computations when used in *INDEP* mode. In this mode, the kernel buffer associated with a *vMAC* stores 16 different kernels (one for each *MAC*). The vector read from the maps buffer is broadcast to all the *MACs* in a pixel by pixel fashion. The values within the vector that are not useful for the computation are discarded. A *CU*, in the *INDEP* mode, produces 64 results in parallel.

In order to achieve an efficient, high speed implementation on FPGAs, the broadcast pixel is selected using a shift register. This results in some wasted cycles, for discarding data within the vector that is unnecessary for a particular computation and causes a minor loss in efficiency. It is notable that despite this loss, Snowflake maintains an average computational efficiency of 91%-95%.

3.4 Data Distribution Network

The data distribution network connects the *CUs*, the Control Core and Memory Interface. It is responsible for loading the operand into the appropriate on chip buffers and storing the results back to memory. It redirects the results of the computation back to the appropriate maps buffer. It also enables the *TMOV* instruction, which allows CUs within a compute core to communicate with each other. The memory interface is AXI4 [20] compliant.

The Control Core, Compute Core and Data Distribution Network together form a Compute Cluster. Figure 2 shows a Compute Cluster and its internals. Snowflake scales at the level of Compute Clusters. Synchronization primitives

are provided for inter cluster coordination. Data movement across clusters currently occurs via main memory.

4 Results

A single cluster Snowflake system was implemented on the Xilinx ZC706. The design was run at 250 MHz and contained 256 MAC units. Total on chip memory was 512 kB with 256 kB each of maps and weights buffers. Snowflake was benchmarked on AlexNet [1], GoogLeNet [2], and ResNet-50 [4]. End to end execution of these benchmarks on Snowflake resulted in average occupancies between 91%-95%. Average bandwidth consumed was about 1.2 GB/s for AlexNet and 3 GB/s for GoogLeNet and ResNet-50. The peak memory bandwidth consumed was 3.3 GB/s for 1×1 convolution layers. For a more detailed view, Figure 3 shows the computational efficiency achieved for these three networks on a per layer basis. The first layer of all networks suffers from an inefficiency due to the implementation of *INDEP* mode explained in Section 3.3. Snowflake is able to achieve 88%-97% efficiency on the various benchmarked layers. A throughput of 98 FPS, 36 FPS and 18 FPS is achieved on AlexNet, GoogLeNet, and ResNet-50 respectively.

5 Conclusion

An efficient, model agnostic CNN accelerator architecture called Snowflake was presented. A 256 MAC Snowflake system was prototyped on a Xilinx ZC706 at 250 MHz. Snowflake was able to achieve average computational efficiencies of 91%-95% on various CNN workloads. The prototyped system was able to deliver real time performance on AlexNet, GoogLeNet and ResNet. Future work involves scaling Snowflake up to a multi cluster system on larger FPGAs. Batch processing will enable Snowflake to maintain the computational efficiencies at current levels. A multi cluster system can be used for server-based workloads where latency is not as important as throughput.

References

[1] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[5] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[6] Z. Wu, C. Shen, and A. van den Hengel, "Wider or deeper: Revisiting the resnet model for visual recognition," *CoRR*, vol. abs/1611.10080, 2016.

[7] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," *arXiv preprint arXiv:1606.01781*, 2016.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[9] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 682–687, 2014.

[10] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[11] 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, no. 1, pp. 127–138, 2017.

[12] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, (New York, NY, USA), pp. 199–204, ACM, 2015.

[13] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pp. 1–4, IEEE, 2017.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *International Conference on Learning Representations (ICLR)*, 2016.

[15] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[16] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1135–1143, 2015.

[17] 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," *International Conference on Computer Architecture (ISCA)*, 2016.

[18] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision.," in *ICML*, pp. 1737–1746, 2015.

[20] A. A. AXI and A. P. Specification-AXI, "Axi4, and axi4-lite, ace and ace-lite," tech. rep., Technical report, 2011.