FPGA-based Accelerator for Losslessly Quantized Convolutional Neural Networks

Mankit Sit, Ryosuke Kazami and Hideharu Amano
Department of Information and Computer Science, Keio University, Yokohama, Japan
Email: asap@am.ics.keio.ac.jp

***Abstract*—Convolutional Neural Networks (CNN) have been widely used for various computer vision tasks. While GPUs are the most common platform for CNN implementation, FPGAs are promising alternatives to provide better energy efficiency. Recent work demonstrates the potential of network quantization to reduce the model size and enhance computation efficiency while maintaining comparable accuracy to the full precision counterparts. Quantized CNN is especially suitable for FPGA implementation due to the presence of values with non-trivial bitwidth. In this paper, we present the design of an FPGA-based accelerator for losslessly quantized CNNs using High Level Synthesis tool. The experiment result shows that our design achieves 12.9 GOPS/Watt for quantized Alexnet on Imagnet Dataset.**

I. Introduction

Convolutional Neural Networks (CNNs) have achieved remarkable success for a wide range of machine learning applications, ranging from image classification to speech recognition, natural language processing etc. CNNs outperform other existing machine learning algorithms in term of accuracy, while demanding enormous computing power and memory capacity. GPUs are currently the most popular solution of CNN acceleration, but they are considered too power hungry in many scenarios. To enable a wider range of applications, especially for embedded computing and datacenter deployment, various hardware accelerators of CNN have been proposed based on FPGAs [1], [2], and ASICs [3], [4]. FPGA-based designs has gained increasing attention because it can achieve better energy efficiency compared to GPUs; it can also provide more flexibility compared to ASICs; more importantly, the recent development of High Level Synthesis (HLS) tools, e.g. Vivado HLS and OpenCL, greatly reduces the development time and programming complexity.

While FPGAs have provided superior energy efficiency than GPUs in CNN acceleration, FPGA implementations can hardly achieve comparable performance to that of GPUs. One of the reason behind this performance gap is that most existing CNN models (e.g. Alexnet [5], VGG-16 [6]) rely on dense computation on floating point data. Compared to GPUs, FPGAs have much fewer floating point units and less on-chip memory buffers, which limits their capability to accelerate the floating point precision models.

To reduce the computational, memory and communication bandwidth requirements, substantial efforts have been made to the compression of CNN models. Some researchers have tried to train CNNs with low-precision weights, activations or even gradients, such as BinaryNet [7], XNOR-net [8] and DeRaFa-net [9]. However, these methods are failed to show competitive accuracy on ImageNet Dataset [10] for compressing deep CNN models (e.g. GoogleNet [11]). Another direction is to quantize a pre-trained CNN full-precision model to lower precision directly. Han et al. [12] have achieved remarkable compression ratio without significant accuracy loss by combining pruning, vector quantization and Huffman coding, which reduces the model size by 35x on Alexnet and 49x on VGG-16. However, the compressed models are still on floating point precision data and are not especially favourable to FPGA-based design.

One possible hardware-friendly quantization scheme is to quantize the model data into power-of-two representation. It has been shown that CNNs can be quantized down to 3 bits without any significant precision loss using logarithmic data representation [13]. Furthermore, a more general quantization scheme called Incremental Network Quantization (INQ) [14] is presented to quantize CNN models down to 4-bit without *any* accuracy drop by groupwise retraining. Under power-of-two representation, most computationally demanding multiplication operations can be converted into efficient bitwise shift operations, which reduces both computation and storage requirement.

In this paper, we present a software/hardware co-design system for accelerating CNN model quantized by INQ method for FPT2017 Design Competition [15]. A design competition held with FPT conference is a tradition to encourage participants to create innovative hardware design for FPGA platforms. Starting from 2017, the topic of the competition is changed from Trax game solver to FPGA-accelerated Deep Learning. Thus, the key challenges of the competition become more focused on hardware architecture instead of algorithmic innovation.

The key design strategies of our work can be summarized as follows:

- Lossless Quantization of Alexnet's model to 5-bit power-of-two representation using INQ to reduce model size and facilitate efficient computation.
- On-the-fly pre-computation of the multiplications between features and filters to allow lookup-table-based multiplication that eliminates most multipliers.
- Utilization of on-chip buffers to store the complete input features as well as partial filter data that reduces the external bandwidth requirement.

978-1-5386-2656-6/17/$31.00 ©2017 IEEE

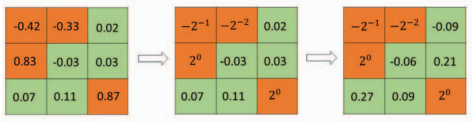

Fig. 1. Incremental Network Quantization

The rest of the paper is organized as follows: Section II provides a background for CNNs and INQ. Section III presents the implementation details of our accelerator architecture. Section IV shows our evaluation result and Section V concludes the paper.

II. Background

A. Convolutional Neural Networks

A convolutional neural network (CNN) is a type of deep learning network commonly used for image processing and computer vision applications. Typically, a CNN consists of convolutional layers, pooling layers and fully-connected layers. Although the topologies and complexities of different CNNs can be varied significantly, they share the common underlying computations that are convolution, activation, max-pooling and matrix-vector multiplication. Among these computations, most of the computational complexity is contributed by the convolution layers and the fully-connected layers. The detail of the three common CNN layers are described below:

1) Convolution Layer: It convolves a 3D input feature map ($N \times H \times W$) with a 4D filter ($M \times N \times K \times K$) to produce a 3D output feature map ($M \times Q \times P$). Specifically, the computation of the layer is defined as:

$$O[m][r][c] = \sum_{n=0}^{N} \sum_{i=0}^{K} \sum_{j=0}^{K} W[m][n][i][j] * I[n][S*r+i][S*c+j]$$

where I, O and W are input feature maps, output feature maps and filters respectively. S is the stride size.

2) Pooling Layer: It subsamples the input feature map by some simple functions; for example, average or maximum. The output feature map usually has a smaller dimension than the input feature map.

3) Fully-Connected (FC) Layer: It takes an input feature vector and multiplies it with a weight matrix to produce an output feature vector. This layer usually appears after all convolutional layer. Essentially, FC layer can be considered as a special form of convolution. The equation below shows the FC layer operation:

$$Out[n] = \sum_{i=0}^{H} W[n][i] * IN[i]$$

B. Incremental Network Quantization

Incremental Network Quantization (INQ) [14] is a lossless compression method to convert any pre-trained CNN model into a compressed version with filters constrained to be power-of-two representation. The general idea of this method is to quantize only a subset of the model weight each time and compensate the accuracy loss due to quantization by re-training.

It consists of three operations: weight partition, group-wise quantization and re-training. A single step of INQ is shown in Fig 1. The weight matrix is partitioned into two disjoint groups. The weights in the first group (highlighted in orange) are quantized into lower precision representation. The weights in the other group (highlighted in green) are then retrained to compensate the accuracy loss from the quantization. These three operations are applied iteratively to the retrained group until all weights are quantized. The quantized CNN models with 5 bits have at least comparable accuracy to their full precision baseline, including Alexnet, VGG-16, GoogleNet and ResNets [16].

An important advantage of this method is that the final quantized weights are in the form of either power of two or zero. Therefore, the original floating-point multiplication operations can be replaced by more hardware-friendly binary bit shift operations, which makes this method particularly suitable for our FPGA implementation.

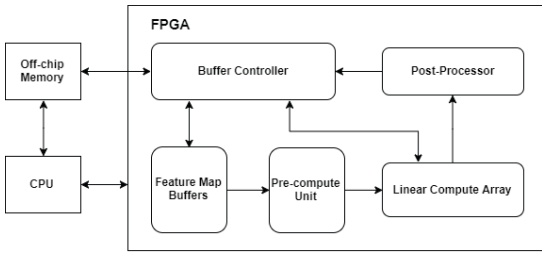

Fig. 2. Overall Architecture of the Accelerator

III. Architecture

A. Overall Architecture

Our accelerator architecture, shown in Fig 2, consists of a buffer controller, feature map buffers, filter caches, precompute unit, compute array and post-processers. The buffer controller manages external data transfers between the global memory and the on-chip buffers (feature map buffers and filter caches) and produces an input feature tile with size $K_{vec} \times C_{vec}$ to the precompute unit each cycle. The precompute unit create a lookup table for all features of the input tile to the compute array. The compute array generates inner products and send it to post-processors for applying ReLU and max-pooling. The post-processors can be bypassed by setting the configuration registers optionally. Finally, the outputs are written back into the feature map buffer and wait for the next layer computation.

B. Filter Data Encoding

An encoded filter data consists of two parts: the sign bit and the index, as shown in Fig 3. The sign bit represents the sign of the quantized filter data. The index indicates the position of the filter data in an ordered quantization set. For example, if the quantization set is $\{2, 0, 2^{-1}, 2^{-2}, 2^{-3}\}$ and the filter data is -2^{-2}, the sign bit and the index are 1 and 3 respectively. The

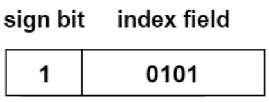

Fig. 3. Filter Encoding Scheme

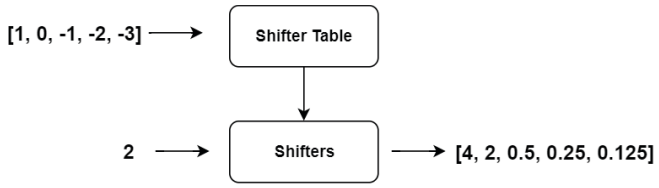

Fig. 4. Example of Pre-Computation

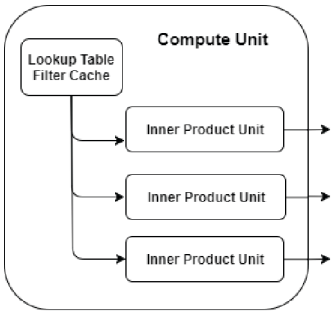

Fig. 5. A Compute Unit

TABLE I
COMPARISON OF THE ACCURACY BETWEEN THE FULL PRECISION ALEXNET MODEL AND THE 5-BIT ALEXNET MODEL

	Top-1 Accuracy	Top-5 Accuracy
Alexnet (5 bits)	55.804%	78.548%
Alexnet (32-bit fixed point)	56.522%	79.066%

encoding scheme facilitates the use of pre-computation and lookup-table-based multiplication in our design which will be explained in the following sections.

C. On-the-fly Pre-Computation

Since all filter data is quantized into a small finite set of values, it is possible to precompute all possible product terms for each of the input features with the quantized filter data.

A precompute unit consists of a block of shifters and a shift table. The multiplication operations between the input features and the quantized filter data are replaced by cheaper bitshift operations. In addition, a shift table is pre-set at the beginning to provide the direction and magnitude of each required shift operation. Each table entry consists of a direction bit and a shift magnitude. The table is configured according to the quantization set. For example, if the quantization set is $\{2, 0, 2^{-1}, 2^{-2}, 2^{-3}\}$, then the table will be set to $\{1, 0, -1, -2, -3\}$ where the sign of each value is the direction bit and the absolute magnitude is the shift magnitude. A feature will be shifted left (right) if the direction bit of the shift table entry is positive (negative). Each feature is shifted with all entries of the shift table.

The operation of the pre-compute unit is shown in Fig 4. Before any computation, each input feature of an input tile enters the precompute unit, is shifted and sent to the compute units. The pre-compute unit can operate on all features of an input tile simultaneously.

D. Linear Compute Array and Lookup-Table-based Multiplication

As shown in Fig 5, P_{vec} compute units are arranged as a linear array which can compute P_{vec} independent output features concurrently. Each compute unit gets one input tile each cycle and produces an inner product per C_{vec} cycles.

The lookup-table-based multiplication is operated in the inner product units of a compute unit. The number of inner product units per compute unit is equal to the size of an input tile.

All inner product units in a compute unit share a lookup table. During operation, an encoded filter data is fetched from filter cache and the index field is used to locate the corresponding lookup table entry. All features of an input tile are then "multiplied" concurrently. The table lookup results are sent to the accumulator to generate a partial sum. The partial sum is either sent out as the final result or stored back to the accumulator registers for next accumulation.

E. On-Chip Buffers

There are two main on-chip buffers in the accelerator. One is feature map buffer and the other is filter cache.

The feature map buffers are used to store the feature data and to send it to the precompute unit. Each buffer is double-buffered which allows concurrent read/write. Before the computation of the first layer starts, feature data are loaded from global memory and stored in the feature map buffers. During the execution of a layer, while the feature data is being sent for convolution, the outputs of convolutions are streamed back into the feature map buffers simultaneously. Therefore, all intermediate values of each layer can be stored in the buffers without any external access in and from the global memory. There are $K_{vec} \times C_{vec}$ feature map buffers which provide the necessary memory ports to fetch one input tile per cycles.

The filter cache is usedto store the encoded filter data for inner products. Each compute unit has its own filter cache. The cache is also double-buffered to overlap the computation and external memory access.

All $P_{vec} \times K_{vec} \times K_{vec} \times C$ filters are partitioned into P_{vec} subsets. At the beginning, a filter subset is loaded from the global memory and is distributed to the filter caches and each filter cache has a single filter tile with size $K_{vec} \times K_{vec} \times C$.

IV. EVALUATION

In this section, we evaluated our accelerator design with Alexnet model on ImageNet Dataset [5]. Model accuracy and accelerator's resource utilization and performance are reported.

TABLE II
PERFORMANCE AND RESOURCE UTILIZATION OF OUR ACCELERATOR AND THE BASELINE

	Our Accelerator	Baseline
LUT	103K (47%)	115K (53%)
FF	275K (73%)	148K (34%)
DSP	108 (12%)	708 (78%)
BRAM	534 (98%)	387 (71%)
GOPS	155.1	103.9
GOPS/Watt	12.9	8.59

A. Model Accuracy

We apply INQ to the Alexnet pre-trained model using Pytorch [17]. We set the INQ's training parameters to the same value in the original paper [14], where the accumulated portion of quantized weight is $\{0.3, 0.6, 0.8, 0.1\}$, the batch size is 256, the weight decay is 0.0005, and the momentum is 0.9. While the number of epochs per INQ iteration for Alexnet is not specified, we use 7 epochs for each finetuning step which can achieve acceptable accuracy with resonable running time. As shown in Table refacc, the pre-trained model is quantized into 5 bit without noticeable accuracy loss.

B. Accelerator

We evaluate the design on a Xilinx Zynq-7000 SoC ZC706 Evaluation board which contains a XC7Z045 SoC, and use Xilinx SDSoC 2017.2 as the primary design tool for high level compilation and FPGA bitstream synthesis. We compare our design with a full precision baseline without any pre-computation. Table II shows the resource utilization of our accelerator and the baseline, and Table II shows throughput and throughput per Watt of our accelerator and the baseline.

1) Resource Utilization: The resource utilization of the baseline and the accelerator implementation is summarized in Table II. It is shown that our design utilizes 30% fewer resources than the baseline. As the design requires much fewer floating point operations compared to the baseline, it requires much fewer DSP blocks. On the contrary, each compute unit requires a lookup table memory to hold the precomputed values, so about 20% more BRAM resources are consumed in total.

As the size of the lookup table is only proportional to the quantization bitwidth and the bitwidth of the input feature, we believe that the design can be scaled up for larger neural network architecutre. We have planned to extend our design for different networks in our future work.

2) Performance: The peak performance of the accelerator is achieved at C_{vec} = 16 and P_{vec} = 32. As shown in Table II, the current best latency is 52.8ms and the peak throughput is 155.1 GOPS and the throughput per Watt is 12.9 GOPS/Watt.

V. CONCLUSION

In this paper, we present an FPGA accelerator for quantized CNN models by INQ. INQ can effectively reduce the model size and convert the model into power-of-two representation, which is well-fitted for hardware implementation. We introduce pre-compute unit in our design to eliminate the expensive multiplications into hardware-friendly bitshift operation, which reduce DSP utilization and improve performance. We use HLS tools to design our accelerator in high level C/C++ code, simplifying the development process. Our design achieves 12.9 GOPS/Watt with less resource utilization.

REFERENCES

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170.

[2] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 55–64.

[3] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 92–104.

[4] 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, Jan 2017.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," pp. 1097–1105, 2012. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[7] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016.

[8] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016.

[9] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "Imagenet: a large-scale hierarchical image database," pp. 248–255, 06 2009.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[13] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *CoRR*, vol. abs/1603.01025, 2016.

[14] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, vol. abs/1702.03044, 2017.

[15] 2017 international conference on field-programmable technology. [Online]. Available: https://www.rmit.edu.au/events/all-events/conferences/2017/december/icfpt2017

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

[17] Pytorch. [Online]. Available: https://github.com/pytorch/pytorch