Efficient Forward Propagation for Binary Deep Neural Network

Utkarsh Singh
University of California, San Diego
A53267107
usingh@ucsd.edu

Swapnil Aggarwal
University of California, San Diego
A53271788
swaggarw@ucsd.edu

***Abstract*—There are many applications scenarios for which the computational performance and memory footprint of the prediction phase of Deep Neural Networks (DNNs) need to be optimized. Binary Deep Neural Networks (BDNNs) have been shown to be an effective way of achieving this objective. Since using binary weights and activations drastically reduces memory footprint and memory accesses, and it could be exploted to replace arithmetic operations with more efficient bit-wise operations, leading to much faster test-time inference and lower power consumption. However, previous works have tried to speed up certain layers in Convolutional Neural Networks (CNNs) but have failed to optimize the same for the entire architecture. In the project, we have implemented the synthesizable LeNet Architecture for BNN's, leveraging bit-packing and bit-wise computations for efficient execution. These techniques provide a speed-up of matrix-multiplication routines, and at the same time, reduce memory usage when storing parameters and activations. A convolutional neural network with binary weights is significantly smaller (32×) than an equivalent network with single-precision weight values. In addition, when weight values are binary, multiply-accumulate operations can be replaced by simple accumulations, which is beneficial because multipliers are the most space and power-hungry components of the digital implementation of neural networks. Binary-weight approximations of large CNNs can fit into the memory of even small, portable devices while maintaining the same level of accuracy. Morover, if both weights and activations are binary, then we could replace multiply-accumulations by the bitwise operations: xnor and bitcount**

***Index Terms*—FPGA, Hardware Acceleration, Binary Deep Neural Network, Binary Convolution, Deep Learning, Convolutional Neural Network,**

I. Introduction

Convolutional neural networks (CNNs) have achieved state-of-the-art results on real-world applications such as image classification and object detection, with the best results obtained with large models and sufficient computation resources. Concurrent to these progresses, the deployment of CNNs on mobile devices for consumer applications is gaining more and more attention, due to the widespread commercial value and the exciting prospect. Thus, improving the test-time performance and reducing hardware costs are likely to be crucial for further progress, as mobile applications usually require real-time, low power consumption and fully embeddable. FPGA has held an outstanding performance in low-power and large-scale parallel computing domain. Compared to CPU or GPU which are based on the Von-Neumann or Harvard Architecture, FPGA has a more flexible framework to implement algorithms. The instructions and data in FPGA can be designed in a more efficient way without the constraint of fixed architectures, which is suitable for designers to explore the high performance implement approaches in power or computing sensitive fields. FPGA with its inherent parallel architecture and other above-mentioned characteristics is best suitable for this task. Our project is about implementing inference of a Binary Neural Network on FPGA.

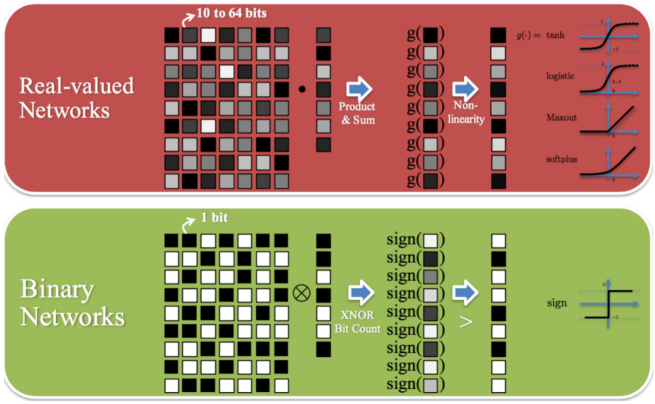

Fig. 1. Comparison of Neural Network [10]

As we could infer from Fig 1, real-valued networks perform convolution by taking in full-precision numbers. The Multiply-add module then calculates the product and sum, apply activation functions and then feed the data to next layer. In contrast, BNNs, however, use +1 and -1 bit values for weights and activation. Apart from the first convolution layer, we perform bit-wise XNOR and then population count. Here, the population count output is in full-precision, however, we perform sign function to again binarise the input which feeds into next layer.

II. Related Works

A. Quantized Neural Networks

High precision parameters are not very necessary to reach high performance in deep neural networks. Recent research efforts have considerably reduced a large amounts of memory requirement and computation complexity by using low bit-width weights and activations. In fact, Zhou et al. [5] and Hubara et al, [4] experiment with different combinations

of bit-width for weights and activations, and show that the performance of their highly quantized networks deteriorates rapidly when the weights and activations are quantized to less than 4-bit numbers.

B. Binarized Neural Networks

The binary representation for deep models is not a new topic. It is known that binary activation can use spiking response for event-based computation and communication (consuming energy only when necessary) and therefore is energy-efficient [6]. Recently, Courbariaux et al. [8] introduce Binarized- Neural-Networks (BNNs), neural networks with binary weights and activations at run-time. Different from their work, Rastegari et al. [7] introduce simple, efficient, and accurate approximations to CNNs by binarizing the weights and even the intermediate representations in CNNs. All these works drastically reduce memory consumption, and replace most arithmetic operations with bit-wise operations, which potentially lead to a substantial increase in power efficiency.

III. Architecture

In order to implement Binarized Neural Networks on hardware, we need an end-to-end system which performs the following tasks:

- Offline Training to obtain binary weights and activations
- HDL interface to encode the hardware functionality using C/C++ or Verilog/System Verilog
- Target Device to implement the network on

For the training part, there are multiple resources available online which provide high level libraries in Python to train the target network on a GPU, in order to obtain the weights and activations. We utilize the training interface from [9], which helps in designing a custom network in tensorflow and then peforming training and dumping of weights in the form of two files: a config file and a params file. The config file contains the various network configuration parameters for each layer, such as for a convolutional layer, the configuration variables can be kernel size, number of input and output channels, data bitwidth, etc. These config files can be imported automatically in Vivado HLS and can be used to design the network in a templatized manner. The training interface also dumps the weights file in the form of parameters. These parameters contain weights and activations defined as arrays of arbitrary precision, in a layer by layer manner. This helps us to effectively read and interpret the code and utilize these parameters easily for model inference.

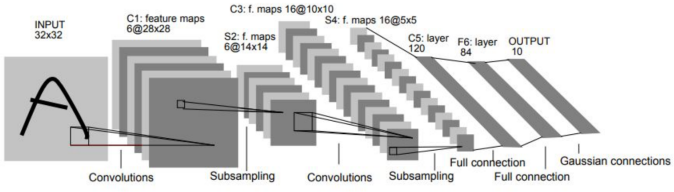

Fig. 2. LeNet-5 Architecture [1]

For the purpose of this project, We implemented the LeNet-5 architecture [1]. LeNet-5 is a simple and popular network which has been developed to train on the MNIST dataset of handwritten digit recognition. It consists of a two convolutional layers, two pooling layers and three fully-connected layers, as specified in Figure 2. Each convolutional layer consists of a 5x5 kernel, which in the original paper performed unpadded convolution on the input feature maps. For the purpose of this project, however, we have used padded convolution in order to preserve the feature map characteristics for better accuracy.

A. Binarization

A BDNN is composed of a sequence of k = 1, . . . , L layers whose weights W^k and activations a^k are binarized to the values $\{1, +1\}$. In order to transform the real valued variables into these two values, we have used sign function to convert the result into two binary value i.e. +1 & -1. As we could infer from Fig 4, Sign function is a deterministic function where $x\hat{b}$ is the binarized variable (weight or activation) and x is the real-valued variable.

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise,} \end{cases}$$

Fig. 3. Signed Binarization

B. Convolution with binary weights

As with fully connected layers with binary weights and activations, we can replace every multiplication with a simple XNOR (equivalence) operation between two bits (i.e. XNOR and Pop Count). Every output map has different parameters, which means that for every output map we have to define a different threshold. In a real implementation, we have to choose carefully which degrees of parallelism to pursue by considering the constraints on our HW resources.

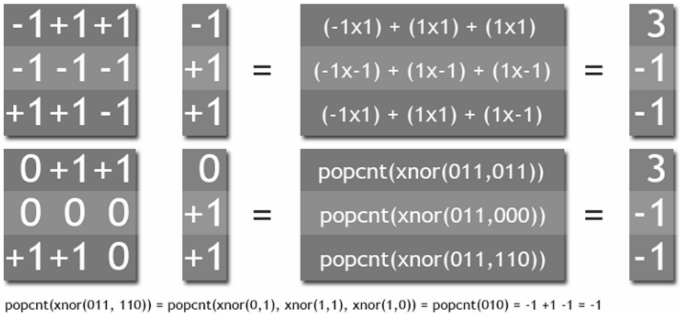

Fig. 4. XNOR Popcount Operation [11]

C. Bit Packing

The weights of a BDNN can be stored in the bits packed using arbitrary precision data values i.e ap_uint 4 which could be used to pack three 1-bit values. We're then using hls stream to stream the packed values from one layer to another, and

finally unroll or unpack them before doing the computation. Bit Packing helps in saving a lot of memory since the space would have not been efficiently utilized if we only saved 1 bit value instead of ap_unit4. One immediate advantage of bit-packing is to drastically reduce the memory usage by a maximum factor of 32×. An even more significant advantage is the ability to process multiple values at the same time using registers. This is particularly useful for dot-products: with bit-packing we can compute a dot-product of 64 element vectors by using just one XNOR and one bit-count.

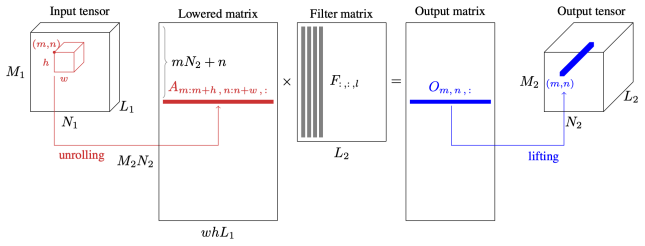

Fig. 5. Unrolling and Lifting in Espresso [2]

IV. Implementation

We have implemented the various layers of LeNet-5 on Vivado HLS - Convolutional layer, Dense Layer and Pooling Layer. In order to fully exploit the various hardware optimizations of HLS, we utilize many HLS-specific pragmas in order to mimic hardware description in C++ code: pragmas such as 'DATAFLOW', 'PIPELINE', 'UNROLL', 'ARRAY_PARTITION, etc. 'DATAFLOW' enables parallel data assignment inside a module function declaration. 'PIPELINE' and 'UNROLL' perform parallel pipelining and instantiation for reducing latency of independent outputs. Moreover, in order to utilize hls streaming using the AXI interface, we need to send the entire MNIST image of 784 (28 X 28) Bytes (or 6272 bits), we cannot use such a large stream size, and therefore we need to break the entire image stream into smaller streams. For this architecture, we chose a stream size of 448b (56B), that translates to 512b (64B) including data check and data keep bits. Therefore, one entire image gets streamed in 14 buffer line outputs (because 448 X 14 = 6272).

In the design process, we initially design one Processing Engine (PE) containing one instance of the inference architecture. We then write a testbench to perform inference on MNIST dataset. Using the training weights, which were calculated as mentioned in Section III, we are able to achieve an average accuracy of around 90-95%.

Afterwards, we design an arbitration scheme for a combination of two PEs. In order to perform this arbitration, we break the input stream of images alternately into two streams, each of which is fed into the output on one PE. Each of the two PEs performs the arbitration in parallel and outputs the predictions to separate output streams, which are then combined alternately in the same order in which they were sent. This design is also first verified on the same testbench in order to make sure we are achieving the same logical functionality.

Fig. 6. Scheduling Results for 2-PE System

V. Results

After we verify the logical functionality, we then synthesize this design for normal and arbitrated output. We keep a default target device (Virtex-7 FPGA board). We first synthesize this design for 1 PE for 2 images and then for 2 PEs and 2 images. Since the 2 PEs are theoretically supposed to run in parallel, we expected the 2 PE system to have almost half the total latency as the 1 PE system. After we synthesize the 1 PE system, we get a maximum clock of 3.717 ns and a total inference latency of 0.416 ms. The 2 PE system, on the other hand, gives the same target clock and a total inference latency of 0.218 ms, which is almost exactly as expected. The latency is not exactly half of 1 PE latency since there exists some pre-bufferring bottleneck for arbiter mechanism of the 2 PE system, whereas we don't need such a mechanism in the 1 PE system. As can be seen from Figure 6, both the PEs work in absolute parallelism and therefore the latency reduces, apart from the arbitration and combination logic.

	Utilization			Timing	
	FIFO	**Total**	**Utilization (%)**	**Clock (ns)**	**Latency (ms)**
1-PE	72	230	11	3.717	43.71
2-PE	1156	1616	78	3.717	21.86

TABLE I
Utilization and Timing Results for LeNet-5

As we scale both the systems upto 100 images for inference, the total latency scales similarly. Finally, the latency for 100 images in inference comes around 43.71 ms for 1-PE system, and 21.86 ms for 2-PE system, finally confirming the successful scalability of the 2-PE system.

Moreover, we achieve a final clock frequency of almost **269 MHz**. Table I specifies the utilization estimates for the 1 PE and 2 PE system. Even with a large input buffer size which can accomodate 10 images at once for each PE, we can fit the entire LeNet on the FPGA board, with a peak utilization of 78%, whereas the 1 PE system consumes just 11%. Note that the extra utilization for 2-PE system here is mostly FIFO buffers, which is necessary to reduce the arbitration latency for 10 images in a buffer. The actual hardware utilization for just the 2-PE system (without the arbiter) is around 25%. Figure 7, 8 and 9 shows the synthesis schematics for convolutional unit, dense unit and 1-PE.

VI. Deliverables

This section outlines our deliverables and our progress on each one of them. In conclusion, we were able to effectively deliver on all of our results with conclusive and positive results. The following points outline the various deliverables and our progress matched for the same:

- **Implement the optimizations mentioned in the paper 'Espresso'**: As elucidated in Section III and IV, we

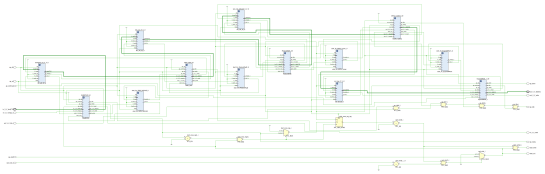

Fig. 7. Synthesis Schematic for Convolutional Layer

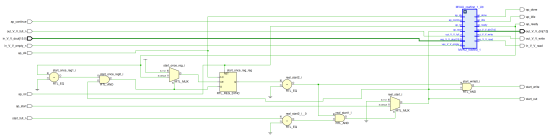

Fig. 8. Synthesis Schematic for Dense Layer

were able to understand and implement all of the bit optimizations such as bit-quantization, bit-packing and unpacking, etc.

- **Design a B-DNN architecture for LeNet-5**: We developed a working architecture for LeNet-5, consisting of 2 convolutional layers, 2 pooling layers and 3 fully connected layers, the last one having a soft max classifier.
- **Synthesis and Implementation on Vivado HLS**: We implemented the entire design on Vivado HLS and generated results for timing and utilization for a given FPGA target (Virtex-7).
- **Check Performance Error on MNIST**: We designed and tested the design on MNIST dataset and achieved a decent accuracy of 90-95 %, which conforms with the standard for MNIST classification using 1 bit weights and activations.
- **Time Multiplexing for Inference**: We replicated the computation for a 2 PE system and performed inference on the same to reduce latency by close to a factor of 2. The various synthesis results have been mentioned in Section V.

VII. Conclusion

We were able to successfully synthesized the binary deep neural network of LeNet architecture through the HLS flow. We achieved three of our key objectives: 1. Attain accuracy of around 95% for the binary deep neural network on MNIST dataset. 2. Achieve maximum clock of 3.717 ns and a total inference latency of 0.416 ms for each PE. 3. Implemented 2 PE's for computation and arbitrated the input stream through them.

As a future work, we would like to automate the whole design flow and arbitrate multiple PE's for increasing the throughput. In addition to it, we would like to add training capabilities, and perform additional performance comparisons on larger standard datasets.

Fig. 9. Synthesis Schematic for 1-PE

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 'Gradient-based Learning Applied to Document Recognition', Proceedings of the IEEE, November 1998.

[2] Fabrizio Pedersoli, George Tzanetakis, Andrea Tagliasacchi, 'Espresso: Efficient Forward Propagation for BCNNs', ICLR 2018.

[3] Vivado HLS Documentation: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf

[4] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, Yoshua Bengio, 'Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations', Journal of Machine Learning Research (2018) pp. 1-30.

[5] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160, 2016.

[6] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, et al. Convolutional networks for fast, energy-efficient neuromorphic computing. Proceedings of the National Academy of Sciences, page 201604850, 2016.

[7] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In European Conference on Computer Vision, pages 525–542. Springer, 2016.

[8] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. arXiv preprint arXiv:1602.02830, 2016

[9] FPGA-based neural network inference for training and implementation: https://github.com/fpgasystems/spooNN

[10] https://software.intel.com/en-us/articles/accelerating-neural-networks-with-binary-arithmetic

[11] https://software.intel.com/en-us/articles/binary-neural-networks