UNIVERSITY OF CALIFORNIA

Santa Barbara

Hardware Acceleration for Tensorized Neural Networks

A Thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

by

Yiming Gan

Committee in charge:

Professor Yuan Xie, Chair

Professor Behrooz Parhami

Professor Li-C Wang

Professor Zheng Zhang

March 2018

The thesis of John Yiming Gan is approved.

_____

Behrooz Parhami

_____

Li-C Wang

_____

Zheng Zhang

_____

Yuan Xie, Committee Chair

March 2018

Hardware Acceleration for Tensorized Neural Networks

by

Yiming Gan

ACKNOWLEDGEMENTS

I want to say thank you to my parents. Without their love and support, it would be impossible for me to finish my master's degree. I also want to express appreciate to all my classmates and advisors in ECE department and Seal lab for their help. I've learned a lot in this two year's process and I'm willing to be better in the following chapter of my life.

ABSTRACT

Hardware Acceleration for Tensorized Neural Networks

by

Yiming Gan

Machine learning has gained success in many application domains including medical data analysis, finance, computer vision, and so forth. However, many popular machine learning models (e.g., deep neural networks) are both data-intensive and computationally expensive: they require high-volume data samples to train the networks, millions to billions of parameters to describe the model, and large-scale computations to complete the optimization or inference. Therefore, deep learning can cause unaffordable energy and run-time cost on a hardware platform. In this paper, we present a way of accelerating deep neural networks as well as compressing weights used by designing hardware acceleration for tensor train decomposition layers in deep neural networks. By utilizing hardware acceleration on tensorized neural networks, we achieved massive memory saving on two fully -connected layers. Parameters shrink 4880644x and 3195660x separately. At the same time, we achieve speed up at 2600x and 2900x compared to original matrix multiplication process.

TABLE OF CONTENTS

I. Introduction and Related Work 1

A. Introduction..... 1

1. Efforts in Hardware Optimization……1

2. Efforts in Algorithm Innovation..…….2

B. Related Work 3

II. Tensor Train Decomposition in Neural Networks..... 4

A. Tensor Train Decomposition 4

B. Tensor Train Decomposition in Neural Networks..... 7

1. TT-layer……5

2. Compute Complexity ……..…….6

III. Hardware Acceleration for Tensorized Neural Networks 8

A. Motivation..... 8

B. FPGA Optimization on TT-layer 8

1. Separate Circuit Design……..10

2. LUT Design for Matrix Reshaping…..……...10

3. Parallel Matrix Multiplication……..……..10

4. Pipeline Different Circuits.……..……...10

C. Evaluation Results..... 11

IV. Discussion and Future Work 13

A. Back-propagation..... 13

B. 3D Process-in-memory 13

References..... 14

I. Introduction and Related Work

A. Introduction

Deep neural network [1] has gained great success at solving a wide range of practical tasks including vision and speech [2], language modeling and translation [3] and autonomous driving [4] . These approaches usually use high-volume of variables to parametrize a neural network, and require high-volume data sets to train the model. For instance, the VGG-19 network needs 500M memory for image recognition [5]. The RNN model in [6] even involves more than 10G parameters. As a result, high-performance computing platforms (e.g., GPU and cloud computing) are usually required to handle the computation and memory expensive training and inference (i.e., prediction) tasks. Training and running a complex deep neural network is infeasible in many application scenarios, due to the limited data sets and computing resources (e.g., power budget and bandwidth constraints on a hardware platform, no access to GPUs or cloud computing) such as IoT (Internet of Things) devices, robotics and embedded platforms. Therefore, developing compact neural network models and designing high performance as well as energy efficient machine learning hardware have become a key enabler for many AI and IoT applications.

1. Efforts in Hardware Optimization.

In the communities of computer architecture and VLSI design, many approaches have been reported to improve the performance of neural network hardware. On GPU and FPGA platforms and in ASIC designs, various techniques have been investigated to reduce both the amount and bit width of the data in weight matrices, neurons and gradients in order to save memory and computation resources [7,8,9,10,11,12]. Customized FPGA/ASIC neural network

accelerators have also been investigated by many groups. For instance, FPGA accelerators have achieved several to dozens of times speedup and energy-efficiency improvement by elaborately designing the parallel processing units, memory allocation and topology tiling. With on-chip DRAM for near-data processing, ASIC accelerators have gained great success. For instance, the DaDianNao[13,14] has gained 450x speedup and 150x energy reduction compared with GPUs by increasing the inner bandwidth of computing parallelism and by reducing the external communications. The performance and energy efficiency of ASIC accelerators have been further boosted by utilizing process-in-memory (PIM) and non-volatile memory-crossbars[15,16,17,18]. For instance, with PIM design, the speed and energy efficiency of the design PRIME[19] are 2360x and 895x better, respectively, compared with GPUs.

2. Efforts in Algorithm Innovation

Meanwhile, the computational math and machine learning communities have proposed some numerical approaches to compress neural networks, but these approaches have been rarely exploited in hardware design. It was first observed that the large weight matrices have low-rank properties, therefore, they could be largely compressed by low-rank matrix factorization[20,21,22]. In Sainish's work, the number of parameters of the network was reduced by 30-50% for a deep neural network for large vocabulary continuous speech recognition tasks. Recently, tensor decompositions were applied to compress neural networks with much higher compression ratios[23,24,25,26]. Tensors are high-dimensional generalization of matrices. While matrix decomposition only compress data in 2 dimensions, tensor decomposition can achieve much higher compression ratio by compressing data arrays in many dimensions. In

the work proposed by Novikov[21] , they first used tensor train decomposition[27] on fully connected(FC) layers and shrink the models for 10^5 times.

B. Related Work

Plenty of works have focused on accelerating deep neural networks from both algorithm perspective and hardware perspective as shown in Fig. 1. In the first work bring tensor train decomposition into deep neural networks, they compressed two FC layers into tensor train format. In the following work, they tried the same method on convolutional layers and get further compression rate. Since fewer parameters are used in presenting layers, the compute complexity is reduced also. The first work designing hardware accelerator for tensor train based deep neural network is proposed by Huang[27], in which they describe a special 3D RRAM and CMOS combined architecture. 3 layers of circuits were used to form the accelerator. The first layer is RRAM-crossbar implemented as a buffer to store weights. Since each tensor core is a 3-D tensor consisted of several slices of matrix, all the matrices are stored in the order of different tensor core. The second layer is another RRAM-crossbar used for computing. It is designed for performing logic operations such as matrix vector multiplication and vector addition. The second layer uses the values stored in the first layer through through-silicon-via(TSV). Layer 3 is designed to perform overall synchronization of the tensorized neural network.

II. Tensor Train Decomposition in Neural Networks

A. Tensor Train Decomposition

Tensors are natural multidimensional generalizations of matrices and have attracted tremendous interest in recent years. Working with d-dimensional problems with d being as

high as 10, 100 or even 1000 has always been nontrivial. Problems of such size cannot be handled by standard numerical methods due to the curse of dimensionality since parameters grows exponentially with d. An efficient representation of a tensor by a small of parameters may give people an opportunity to work with high dimensional problems. There are several widely used tensor decomposition method including canonical decomposition, Tucker decomposition and tensor train decomposition. A d-dimensional $n_1 \times n_2 \times \ldots \times n_d$ tensor A is said to be in the TT-format with cores G_k of size $r_{k-1} \times n_k \times r_k, k = 1,2, \ldots, d$ and $r_0 = r_1 = 1$, if its elements are defined by formula(2.1).

(2.1) $$\mathrm{A}(i_1, \ldots i_d) = \sum_{\alpha_0, \ldots, \alpha_d} G_1(\alpha_0, i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \ldots G_1(\alpha_{d-1}, i_d, \alpha_d)$$

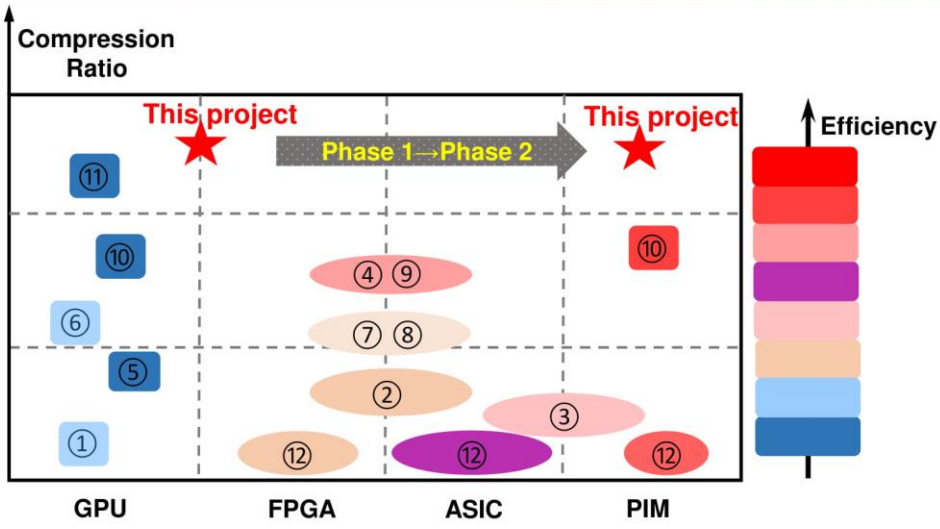
In which three-dimensional tensors G_k are called cores of the TT-decomposition and are all needed to be stored in memory. The ranks r_k are called compression ranks or TT-ranks. By decomposing tensors into tensor train format, basic tensor operations such as addition and Hadamard product can be performed based on tensor cores shown in table 1.

Table 1. Tensor operations in tensor train format

Operation	Formula	Tensor Cores
Addition	$C_{i_1 \ldots i_d} = A_{i_1 \ldots i_d} + B_{i_1 \ldots i_d}$	$C_k = \begin{pmatrix} A_k & 0 \\ 0 & B_k \end{pmatrix}$
Hadamard product	$C_{i_1 \ldots i_d} = A_{i_1 \ldots i_d} B_{i_1 \ldots i_d}$	$C_k = (A_k \otimes) B_k$

Figure 1. Existing software and hardware solutions for deep neural networks

Index	Existing Implementations	Compression Method	Hardware Platforms	Inference	Training
①	Mixed-precision	Quantization	GPU	√	√
②	ReLU-Zeros	Zero Compression	FPGA, ASIC	√	×
③	16b/8b fixed-point	Quantization	ASIC, PIM	√	×
④	Deep Compression	Pruning	FPGA, ASIC	√	×
⑤	Structural Sparsity	Pruning	GPU	√	×
⑥	Hashing Compression	Pruning	GPU	√	√
⑦	SVD Decomposition	Approximation	FPGA, ASIC	√	×
⑧	Huffman Encoding	Approximation	FPGA, ASIC	√	×
⑨	Weight Share	Quantization	FPGA, ASIC	√	×
⑩	Binary/Ternary NNs	Quantization	GPU, PIM	√	×
⑪	Partial Tensorizing	Approximation	GPU (no-optimization)	√	√
⑫	Accelerators	No-compression	FPGA, ASIC, PIM	√	×/√



B. Tensor Train Decomposition in Neural Networks

1. TT-layer

The direct application of the TT-decomposition to a matrix (2-dimensional tensor) coincides with the low-rank matrix format. A matrix in the TT-format is called TT-matrix and a TT-matrix is not restricted to be square. All operations available for TT-tensors are applicable to the TT-matrices as well. TT-matrices can perform matrix-by-vector(matrix-by-

matrix) easily which will be described later. If one of the operands is in TT-format, the result would be a vector(matrix). If both operands are in the TT-format, the result would be in TT-format with increasing ranks on TT-cores.

When bringing tensor train decomposition into neural networks, TT-layer is introduced. In this work, fully-connected layers are used as an example to explain how tensor train decomposition are used in neural networks. As for other layers such as convolutional layers, they can be transformed into the same compute format with FC layers. Fully connected layers apply a linear transformation to an N-dimensional input x:

$$y = Wx + b \tag{2.2}$$

in which the weight matrix W and the bias vector defines the transformation.

A TT-layer is the fully-connected layer in which weight matrix W is stored in TT-format allowing to store only tensor train cores instead of original weight matrix which save the parameters in a low rank way. The actual parameters stored can be controlled by changing tensor core size. The linear transformation shown in (2.2) can be expressed in the tensor form:

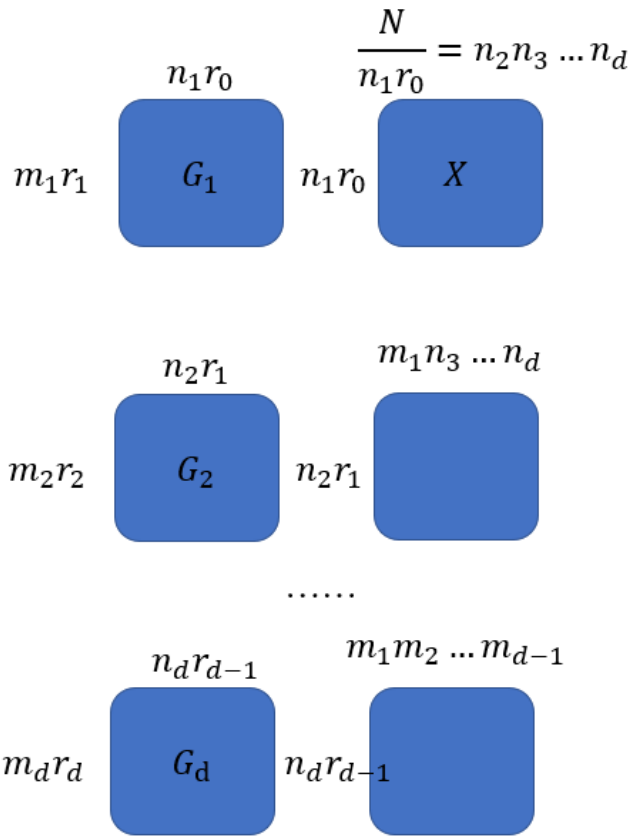
$$y(i_1, i_2, \dots, i_d) = \sum_{j_1, j_2, \dots j_d} G_1 G_2 G_3 \dots G_d \times x(j_1, j_2, \dots, j_d) + b(i_1, i_2, \dots, i_d) \tag{2.3}$$

Since the process is transformed from matrix multiplication to matrix times tensor cores, the dimensional matching becomes a problem. How data is represented and multiplied will be shown in next part.

2. Compute Complexity

When the weight matrix is represented in TT-format, original matrix multiplication become tensor matrix multiplication. The process is shown in figure 2 in which $N = n_1 n_2 n_3 \dots n_d$ and $M = m_1 m_2 m_3 \dots m_d$.

Figure 2. TT-layer multiplication process



The first tensor core will be unfolded to a matrix and reshaped to the matching size with reshaped input matrix x. The first sub-process is a matrix multiplication with the complexity of $C = m_1 r_1 n_1 r_0 n_2 \dots n_d = r^2 m_1 n_1 \dots n_d$ resulting an output matrix in size of $n_2 r_1 \times m_1 n_3 \dots n_d$. The following tensor core is then reshaped to the matching size with previous output and performs the second matrix multiplication with the complexity of $C = m_2 r_2 n_2 r_1 n_3 \dots n_d = r^2 m_1 m_2 n_2 \dots n_d$. There will be d iterations in total and the final compute complexity is $O(dr^2 m \max\{M,N\})$. Compared to the previous fully-connected layer, TT-layer doesn't only save memory on storing parameters but also reduce the compute complexity from O(MN) to $O(dr^2 m \max\{M,N\})$.

III. Hardware Acceleration for Tensorized Neural Networks

A. Motivation

Although it has been proved that using tensor train decomposition on compressing deep neural networks will not only achieve obviously compression rate, but also have better compute complexity since weights represented in tensor core format are much smaller than before. However, from the data of our profiling, the real performance of tensor based deep neural networks seems worse. As shown in Fig.3, we did profiling based on comparison of the baseline which is VGG-16 and compressed networks, another comparison has been made between two simple networks with one TT layer plus one FC layer and two FC layers. The result showing in Table 2 and 3 prove that although theoretically compute complexity will be reduced, tensorized neural networks don't show actual performance improvement. The evaluation platform is Intel® Xeon® E5-2603 CPU with a Nvidia® Titan XP GPU.

The reasons for the profiling results are not hard to explain. In actual operations, weights are stored in memory. Each time one matrix multiplication is processed, the weights will be taken from memory and be put into GPU for computation. In tensor train computation, each tensor core will be multiplied by a matrix. To complete this multiplication, some reshape procedure will be required as mentioned in the previous section. Although compute complexity is largely reduced, the time that reshape is happening is not taken into consideration. That's the reason that tensorized neural networks will take longer time than it supposed to and why hardware acceleration is needed for this kind of networks.

B. FPGA Optimization for TT-layer

FPGA is a popular option of existing accelerators, and it is widely used in embedded environments and in cloud servers. FPGA consists of hundreds of thousands of programmable logic blocks and programmable interconnections, which can provide reconfigurable functions; FPGA also consumes less power than GPU, therefore, it can be utilized in embedded scenarios. Four methods have been used in this paper on FPGA optimization for TT-layer.

Figure 3. Profiling on networks including TT-layer

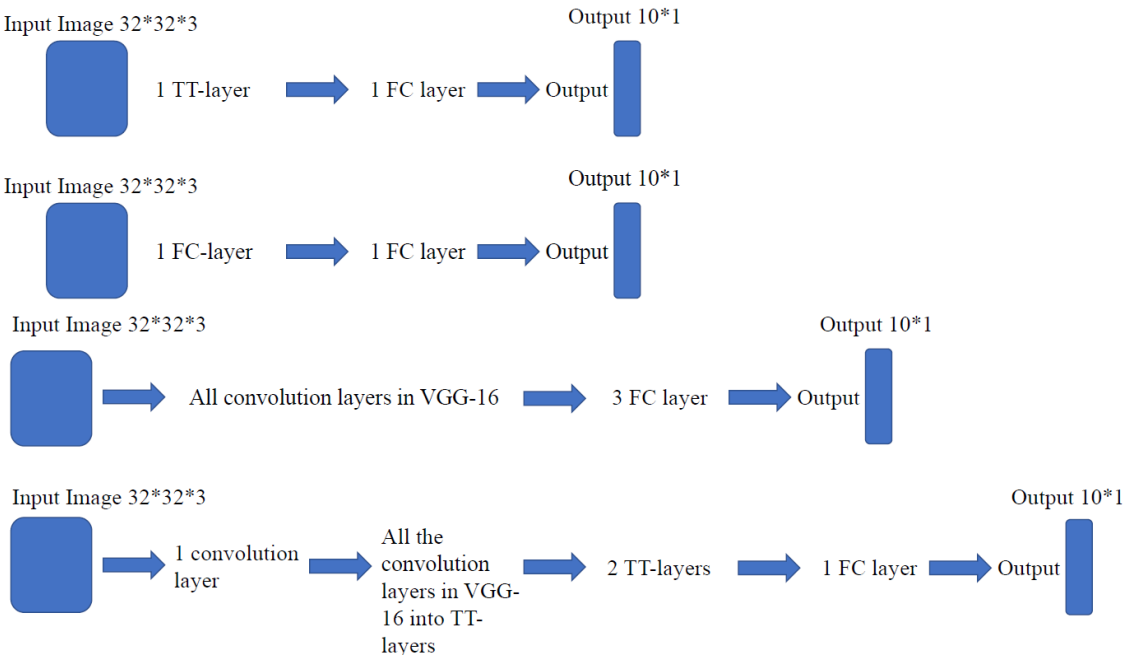


Table 2. Profiling Results for TT-layer

Networks	2 FC layers	1 FC layer+1 TT-layer
Runtime(ms)	11	57

Table 3. Profiling Results for Tensorized VGG-16

Networks	VGG-16	Tensorized VGG-16
Runtime(ms)	132	127

1. Separate Circuit Design

As shown in the previous section, every iteration dealing with TT-layer consists of several different matrix multiplication and reshaping. Since once circuits are designed and build, the data set size it can deal with (i.e. matrix size) can not change, we design separate circuits for different parts in one iteration. Each circuit will be designed to handle one matrix multiplication and its related reshape process.

2. LUT Design for Matrix Reshape

Look-up table method is used when performing matrix reshape. All the elements are stored in a 1-d array and each element will have a corresponding index after reshaping under the reshaping rules (i.e. 4×4 to 8×2). This is a perfect look-up table problem that can easily be solved on FPGA. Corresponding calculation is the only time-consuming part that can be done in parallel since each element's position after reshaping is independent.

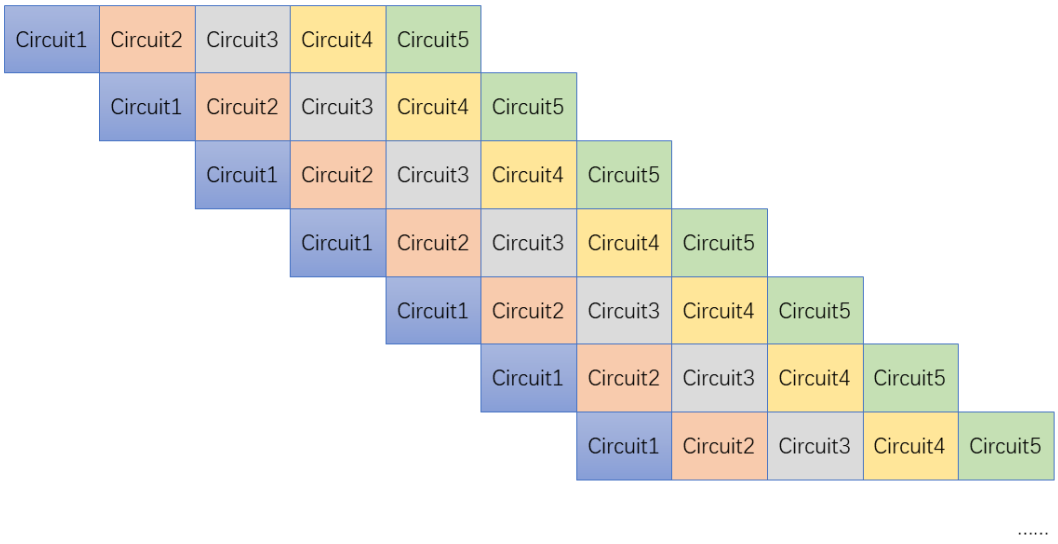
3. Parallel Matrix Multiplication

Matrix multiplication parallelization has been studied a lot on both FPGA and GPU. We also included some basic parallel optimization on matrix multiplication. Assuming a matrix multiplication happens between two matrices of size $n \times n$, each resulting element requires n times multiplication and addition which can be done in 2 clock cycles in the most extreme parallelization case. As a result, it requires n multiplier and a n-input adder. We also included block matrix multiplication in the process.

4. Pipeline Different Circuits.

This optimization is for multiple iterations. Since there will be no dependencies between second iteration and first iteration, a pipeline design shown in Figure 4 can be achieved.

Figure 4. Pipeline for multiple iterations



C. Evaluation Results

We ran the evaluation using High-Level-Synthesis(HLS) solution from Xilinx. We implemented on Xilinx Vivado HLS software and our targeting platform is Zynq xc7z100 FPGA. We implemented two very large fully-connected layers in tensor train format, the first weight matrix is in the size of 3072×262144 and the second one is 262144×4096, the input vector size is 1×3072. Each matrix is represented by 6 tensor cores, thus 12 circuits are needed for these two TT-layers. As shown in Figure 5, orange bars represent each circuit's runtime performance before optimization, blue bars represent the result after optimization. On average 24x speed up can be achieved due to our FPGA optimized methods. Figure 6.a and Figure 6.b shows the overall comparison between optimized TT-

layer with original matrix multiplication in fully-connected layer. TT-layers have shown 148x and 128x speed up already, when fpga optimization was used, the overall speedup have increased to 2643x and 2944x respectively. The total resource used in 12 circuits are shown in Table 4. The memory usage are reduced 4880644x and 3195660x respectively.

Figure 5. FPGA optimization results

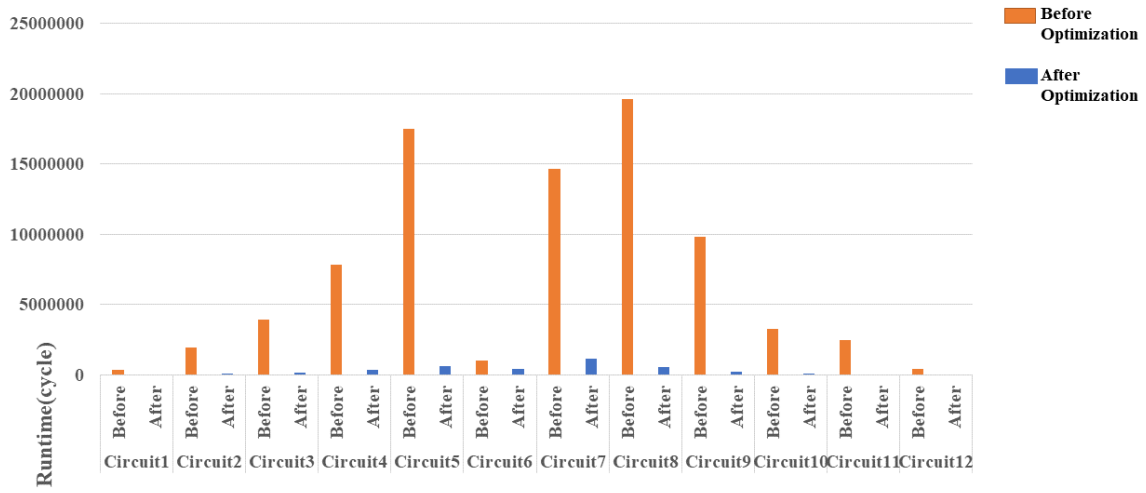


Figure 6. Overall Optimization Results

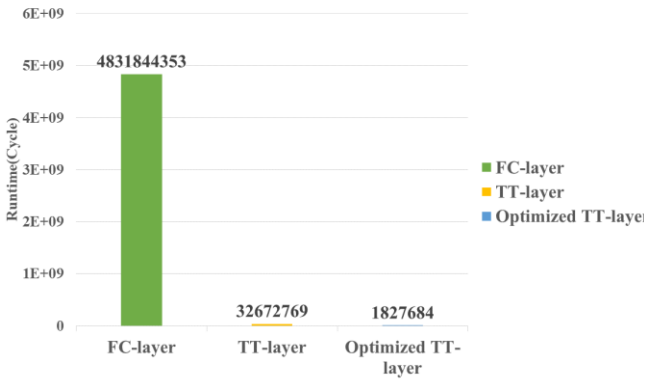


Figure 6.a

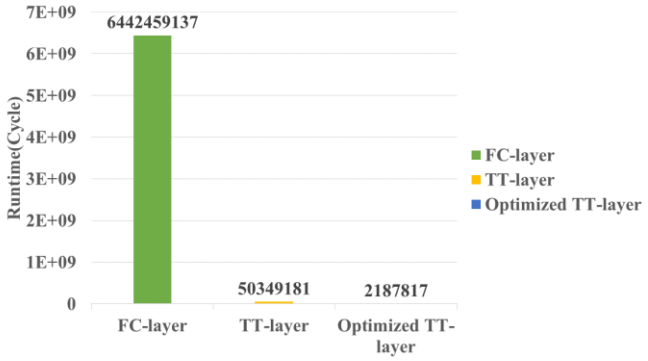


Figure 6.b

III. Discussion and Future Work

A. Back-Propagation

Neural networks are usually trained with the stochastic gradient descent algorithm where the gradient is computed using the back-propagation procedure. Back-propagation allows to compute the gradient of a loss-function L with respect to all the parameters of the networks. Our method optimized the performance of forward pass, but the backward pass compute complexity is also improved because of TT-layer. Back-propagation requires different operations which means different fpga optimization are needed to accelerate it. A complete hardware accelerator needs to include both forward-propagation and back-propagation.

B. 3D Processing-in-memory

Processing-in-memory(PIM) was first proposed in 1990s to address the memory-wall issue on von-Neumann architecture. Emerging technology of 3D die stacking has mitigated the manufacturing issues of PIM to some extent. The vertical 3D die-stacking with TSVs allows stacking multiple memory dies directly on top of the logic die to achieve high memory bandwidth. Since tensor train decomposition represents data in 3-dimensional tensor cores, it perfect matches the characteristic of 3D PIM which leaves room of imagination that using 3D PIM can solve the redundant unfolding and reshaping in neural network process.

Table 4. FPGA Resource Utilization

Resource	DSP48E	Flip Flop	LUT
Utilization	33	36625	88723

References

1. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature,* vol. 521, no. 7553, pp. 436-444, 2015.

2. L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *Acoustics, Speech and Signal Processing(ICASSP), 2013 IEEE International Conference on,* pp. 8599-8603, 2013.

3. R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of 25th international conference on Machine Learning,* pp. 160-167, 2008.

4. C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision,* pp. 2722-2730, 2015.

5. K. Simonyan and A. Zisserman, "Very deep convolutionial networks for large-scale image recognition," *arXiv preprint arXiv: 1409.1556,* 2014.

6. A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep speech 2: End-to-end speech recognition in English and mandarin," in *International Conference on Machine Learning,* pp. 1337-1345, 2013.

7. 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,* pp. 1135-1143, 2015.

8. S. Han, H. Mao and W.J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *arXiv preprint arXiv: 1510.00149,* 2015.

9. S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, *et al.,* "Dsd: Dense-sparse-dense training for deep neural networks," 2016.

10. S. Narang, G. Diamos, S. Sengupta, and E. Elsen, "Exploring sparsity in recurrent neural networks," *arXiv preprint arXiv:1704.05119,* 2017.

11. S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al. "*Ese: Efficient speech recognition engine with sparse lstm on fpga.,*"* in *FPGA,* pp. 75-84, 2017.

12. J. Ba and B. Frey, "Adaptive dropout for training deep neural networks," in *Advances in Neural Information Processing Systems,* pp. 3084-3092, 2013.

13. 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," in ACM Sigplan Notices, vol. 49, pp. 269-284, ACM, 2014.

14. Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*," Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEE/ACM International Symposium on Microarchitecture,* pp. 609-622, IEEE Computer Society, 2014.

15. S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conference(DAC), 2016 53rd ACM/EDAC/IEEE,* pp. 1-6, IEEE, 2016.

16. D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory, " in *Computer Architecture(ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on,* pp. 380-392, IEEE, 2016.

17. S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govingdaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, *et al.*, "A many-core architecture for in-memory data processing, " in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 245-258, ACM, 2017.

18. M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 751-764, ACM, 2017.

19. P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture,* pp. 27-39, IEEE Press, 2016.

20. T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets, " in *Acoustics, Speech and Signal Processing(ICASSP), 2014 IEEE International Comference on,* pp. 6655-6659, 2013.

21. Y. Zhang, E. Chuangsuwanich, and J. Glass, "Extracting deep neural network bottleneck features using low-rank matrix factorization," in *Acoustics, Speech and Signal Processing(ICASSP), 2014 IEEE International Conference on,* pp. 185-189, 2014.

22. J. Xue, J. Li, D. Yu, M. Seltzer, and Y. Gong, "Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network," in *Acoustics, Speech and Signal Processing(ICASSP), 2014 IEEE International Conference on,* pp. 6359-6363, 2014.

23. A. Novikov, D. Podoprikhin, A. Osokin, and D.P. Vector, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems 28,* pp.442-450, 2015.

24. V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," *arXiv preprint arXiv: 1412.6533,* 2014.

25. Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *Proceedings of the 34th International Conference on Machine Learning,* vol. 70 of *Proceedings of Machine Learning Research,* pp. 3891-3900, PMLR, 06-11 Aug 2017.

26. A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," *CoRR,* vol. abs/1705.08052, 2017.

27. H. Huang, L. Ni, K. Wang, Y. Wang and H. Yu, "A highly-parallel and energy efficient 3d multi-layer cmos-rram accelerator for tensorized neural network," *IEEE Transactions on Nanotechnology,* vol. PP, no. 99, pp. 1-1, 2017.