Sparse Ternary Connect: Convolutional Neural Networks Using Ternarized Weights with Enhanced Sparsity

Canran Jin, Heming Sun and Shinji Kimura

Graduate School of Information, Production and Systems, Waseda University
2-7 Hibikino, Wakamatsu-ku, Kitakyushu, Fukuoka 808-0135, Japan
jincanran@fuji.waseda.jp

Abstract— Convolutional Neural Networks (CNNs) are indispensable in a wide range of tasks to achieve state-of-the-art results. In this work, we exploit ternary weights in both inference and training of CNNs and further propose Sparse Ternary Connect (STC) where kernel weights in float value are converted to 1, -1 and 0 based on a new conversion rule with the controlled ratio of 0. STC can save hardware resource a lot with small degradation of precision. The experimental evaluation on 2 popular datasets (CIFAR-10 and SVHN) shows that the proposed method can reduce resource utilization (by 28.9% of LUT, 25.3% of FF, 97.5% of DSP and 88.7% of BRAM on Xilinx Kintex-7 FPGA) with less than 0.5% accuracy loss.

I. Introduction

With the rapid development of Convolutional Neural Networks (CNNs), we can achieve the state-of-the-art results in various tasks, such as computer vision and artificial intelligence. Over the past few years, there is a trend to deploy CNNs on CPUs or GPUs because of their ability in parallel computing. Many excellent software frameworks are developed such as Theano [1], TensorFlow [2] and Torch [3]. These frameworks facilitate the application of the latest models and the tuning of custom networks, which greatly promote the researches on deep neural networks.

Recently, the model structures of CNNs become deeper and deeper to achieve higher classification accuracy with large training sets [4], [5], which brings great challenges in computation and storage capacity to conventional accelerators. Since most of computation in a conventional convolutional neural network is floating point multiply-and-accumulate operation, several researchers have proposed some precision-reduced approach to simplify computation. Binary Connect (BC) [6] is a computation method that constraints weights to only two possible values (i.e. -1 or 1)in training while retaining the precision of the weights for gradient descent calculation. BC eliminates the multiplications during forward propagation and backward propagation and even helps CNN models to generalize better as a regularizer. Binary weights are also exploited to be used in classification inference and proved to be effective in complexity reduction of networks while still achieving high accuracy [7], [8]. Based on BC, Ternary Connect (TC) [9] is proposed allowing weights to be zero, which further improves the model performance and retains the efficiency in computation.

A few researchers introduce these quantized-network methods into actual hardware design [10]. In this paper we focus on exploiting the ternary-weight method in hardware accelerators due to its superiority in performance and exploitable sparsity.Since floating point multiplications can be greatly eliminated by using ternary weights, the floating point additions remains become the bottleneck for low-power and resource-efficient hardware design. A certain amount of zero-weight is unevenly distributed on convolutional kernels, which brings difficulty to benefit from zero-weights in aspect of hardware. We introduce the enhanced sparsity into TC method to make zero-weights evenly spread across kernels and further reduce the floating point additions in convolutional layers for shrinking the area of convolutional computation units in actual hardware implementation. The contributions of this paper are as follows:

1) Sparse Ternary Connect (STC) is proposed, which is an enhanced method to improve the sparsity in ternary networks towards hardware design.

2) Verification and demonstration using Theano show that the proposed STC method can be used in both training and inference phase and increases the convolutional layer sparsity from 20.07% to 44.78% with slight accuracy loss compared to original TC.

3) We also design and implement the preliminary TC and STC architectures on FPGA to evaluate the proposed STC and show that the method can save considerable resource utilization with respect to both TC and conventional implementations.

The rest of this paper is organized as follows: Section 2 introduces related works. The proposed STC and its architecture are described in Section 3 and 4. Section 5 presents the experimental results and the final section is the conclusion and future work.

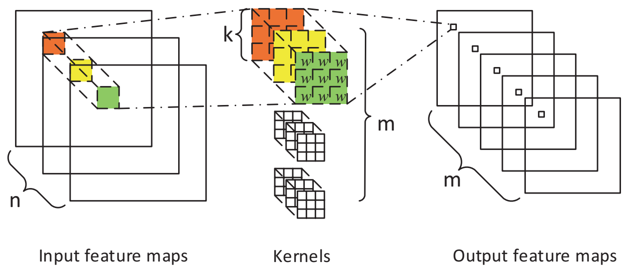


Fig. 1. Computation in a typical convolutional layer with $k \times k$ kernels.

II. Related works

A. Convolutional neural networks

A typical CNN consists of convolutional layers, pooling layers, and fully connected layers. The training process consists of two phases: the forward propagation (FP) and backward propagation (BP). In FP phase, given the CNN input, CNNs compute the unit activations from input to output layer. The BP phase calculates how much those weights contribute to the deviation from the target value and then updates them.

In normal training conditions, we should do FP and BP repeatedly to make sure that the neural networks with stochastic initial parameters converge enough to optimal solution. When trained networks are deployed for specific tasks (e.g. image classification, face recognition), only FP phase is needed to perform inference.

As most of the computation and energy consumption are concentrated on the convolutional layers, we focus on the convolutional layers. 2D convolution is computed as:

$$conv2D(W, x, u, v) = \sum_{i=1}^{k}\sum_{j=1}^{k} W(i,j)x(u+i, v+j) \quad (1)$$

where W is a $k \times k$ weight matrix, x, u, v are an input feature map and the coordinates on it. Convolutional layers accept n input feature maps to produce m output feature maps and extract local features of images to generate feature maps by using the m number of different kernels with n depth. Figure 1 illustrates the computation in a typical convolutional layer.

B. Ternary Connect

As discussed in Section I, this work is based on TC. TC is a computation method to eliminate most of multiplications in the training phase for neural networks. In stochastic gradient descent (SGD), sufficient precision is necessary to accumulate and average a large number of stochastic gradients. However noisy weights are quite compatible during training. Each time when convolution operation is executed, TC firstly clips the weights in the interval $[-1, 1]$, then generates stochastic weights in place of the real-value weights.

TC uses a real weight $W \in [-1, 1]$ as the probability to convert the weight to the ternary weight $W_t \in \{-1, 0, 1\}$. If $W \geq 0$,

$$P(W_t = 1) = W; P(W_t = 0) = 1 - W \quad (2)$$

if $W < 0$,

$$P(W_t = -1) = -W; P(W_t = 0) = 1 + W \quad (3)$$

where $P()$ is the probability, W_t is the ternarized weight and W is the real-value weight. For each weight in stochastic ternary kernel, the closer its corresponding real weight is to zero, the higher probability it will get to become zero.

In TC, weights are stochastically ternarized. In this way, the complex floating point multiplications become sign changes and multiply-accumulate operations are replaced by simple accumulations. It significantly reduces the amount of computation by eliminating floating-point multiplications during training.

At first, we designed a ternary-weight processor by replacing the multipliers by multiplexers. The synthesis result shows that after reducing the dependence on multiplier, the floating adders consume most of the resource (49.6% of LUTs) in core computation modules. Therefore, we further propose a method to exploit the sparsity in ternarized weights for reducing the use of floating adders.

III. Proposed Sparse Ternary Connect

A. Ternary weights with enhanced sparsity

In the original TC, weights can be -1, 0 and 1 and indefinite number of weights will be assigned to zero. If the weights are zero, the corresponding signals need not to be accumulated. However, the original ternarized method can not directly relate to area reduction. So we propose STC, an improved method to explore the potential of TC and further reduce the utilization of floating adders.

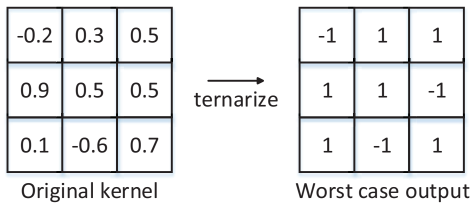
The objective of STC is to enforce definite number of zero weights in filter during ternarization so that a fixed number of additions can be skipped in each convolution operation. Accordingly, the number of adders in convolution computation units may be significantly reduced. To do so, a parameter ρ $(0 \leq \rho \leq 1)$ is introduced, and $N = |W| * \rho$ weights are set to zero where $|W|$ is the size of kernel weights.

Figure 2 illustrates the difference between TC and STC conversions. In the original TC, considering the worst case, there is no zero in a converted matrix, therefore 8 adders are necessary to accumulate the 9 element-wise product. As for the proposed STC with $\rho = 0.5$, there are at least 4 zero elements existing in matrix, only 4 adders are needed for the accumulation.

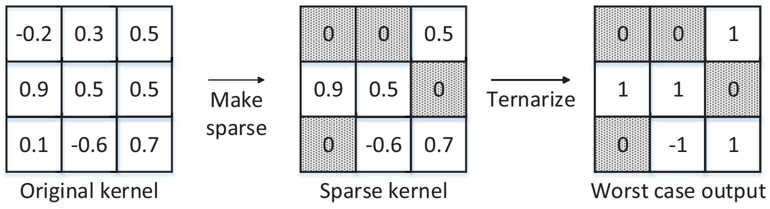
Algorithm 1 is the pseudo code to show how STC works in the forward and backward propagation. First, the corresponding real-value weight matrixes are converted to sparse ternary weight matrixes. Then the convolution operations are conducted using the converted filters. Particularly, the weights should be kept in sufficient precision when updating.

The computation flow of $sparse\_ternarize(W, N)$ function can be summarized as following 3 steps:

Step 1 Given a customized sparsity parameter ρ, calculate the target number of weights N which need to be zeroed.



(a) Example of original ternary connect in the worst case (no zero in converted matrix).



(b) Example of proposed sparse ternary connect in the worst case. When $\rho = 0.5$, 4 elements in a 3*3 kernel are zeroed.

Fig. 2. Process of the origianl TC and the proposed STC (in $3 * 3$ kernel case).

Algorithm 1 STC in typical convolutional layer l, which has n input channels and m output channels. C is the cost function and the function $clip\,(w)$ clips weights into $[-1, 1]$.

```
Require: kernels $W$, bias $b$ in this layer. Layer input $x$, learning rate $\eta$, customized sparsity parameter $\rho$.
Ensure: update $W$ and $b$.
    Forward propagation:
 1: for each kernel $W$ in $range\,(1, m * n)$ do
 2:     $N \leftarrow |W| * \rho$
 3:     $W_{s,t} \leftarrow sparse\_ternarize(W, N)$
 4: end for
 5: Compute convolutional output $y$ knowing $x$ , $W_{s,t}$ and $b$
    Backward propagation:
 6: After getting error signal $\delta_{l+1}$ propagate from its next layer
 7: Compute $\delta_l = \frac{\partial C}{\partial y}$ knowing $\delta_{l+1}$ and $W_{s,t}$
 8: Compute $\frac{\partial C}{\partial W_{s,t}}$ and $\frac{\partial C}{\partial b}$ knowing $\delta_{l+1}$ and $y$
 9: Update $W$: $W \leftarrow clip(W - \eta\frac{\partial C}{\partial W_{s,t}})$
10: Update $b$: $b \leftarrow b - \eta\frac{\partial C}{\partial b}$
```

Step 2 Get the positions of N weights closest to zero in filter matrix.

Step 3 Obtain the sparse kernel by setting these N weights to be zero and making weights ternary.

Note that positions of zero depends on a kernel, so when constructing accumulation module a selector is necessary to select non-zero parts. The cost should be smaller than that of erased adders.

B. Inference-oriented strategy

In the original TC, the real-value weights are ternarized in a stochastic way as shown in Eq.2 and Eq.3. It is meaningless to perform stochastic ternarization in STC when performing inference, because the stochastic sampling will lead to an uncertain result. It is also not efficient to equip stochastic number generators (SNG) for each weight element when designing the customized hardware. In order to use sparse-ternary weights in test-time inference, we recommend that deterministically decide the ternarized weights of a trained network. Thus we proposed an optimized deterministic ternarization as the following equations:

$$W_t = \begin{cases} 1 & if\ W \geq \dfrac{1}{3}, \\ 0 & otherwise, \\ -1 & if\ W \leq -\dfrac{1}{3}. \end{cases} \tag{4}$$

Ternarization in this way can be easily executed by hardware without SNG and greatly benefit the inference process on hardware.

The comparison between the stochastic and deterministic binarization has been discussed in BC [6]. In TC, the deterministic method can play a similar role, which means the method leads to a slightly worse but acceptable performance in averaging process and using the ternary network trained by the same deterministic form of ternarization to perform ternarized inference also makes the most sense.

In this study, deterministically ternarized weights are used for computing the parameter gradients during training as well as for producing network outputs at test-time inference.

C. Extension to training

The proposed STC is able to be only used in training phase while we use real-value weights at test-time (i.e. inference at run-time). When concerning about the training performance only, the stochastic form of ternarization performs better to collect gradients compared to the deterministic one. We could expect from TC that the proposed method may act as a regularizer to avoid overfitting.

Therefore, in our experiment, we use the stochastic form of ternarization in training and the real-value weights in inference phase, which helps to show the effect of STC in training intuitively (the same as that in BC [6] and TC [9]).

IV. Hardware Implementation

We preliminarily implemented the proposed STC method by altering a present FPGA-based CNN processor architecture to verify the corresponding improvement in performance and resource effectiveness. Since the convolution operations in CNN accelerators are typical multiply-and-accumulate processes, the proposed method can be easily applied in basic architecture.

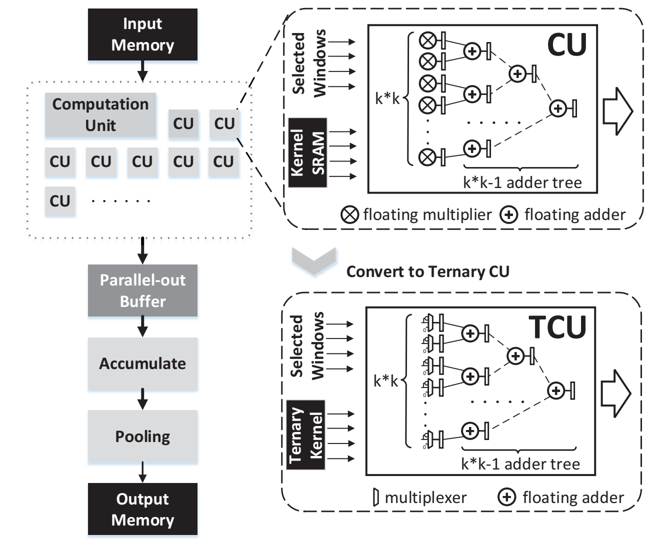


Fig. 3. Basic architecture for CNN processors. Computation units for $k \times k$ kernel. The parallelism of CUs determines the throughput of processors.

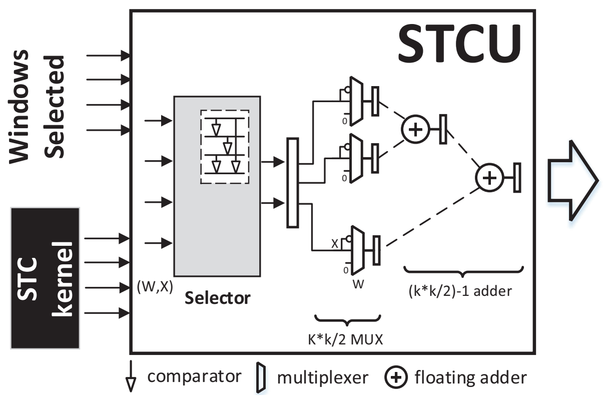


Fig. 4. Architecture for sparse-ternary CU when $\rho = 0.5$, half elements in weight matrix are dropped. Here, the kernel size is $k \times k$, in our implementation the kernel size is 3×3.

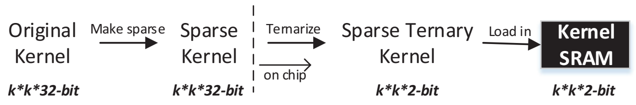


Fig. 5. Bit-width change of kernels, memory bandwidth may further reduced if kernels ternarized off-chip.

A. Basic system architecture

Figure 3 shows the overview of a typical CNN processor and its basic implementation of convolutional computation units (CUs). Similar to CNN-MERP [11], this architecture places an array of convolutional CUs in parallel. All kernels are stored in SRAM on chip in advance and input windows are reused between filters to reduce input memory traffic. The result of each CU is accumulated into a parallel-out buffer preparing to be processed at the following stage. In CUs, $k \times k$ floating-point multipliers are placed in parallel, with $k \times k - 1$ floating-point adders followed.

B. Ternary computation unit

According to the characteristic of Ternary Connect, we design the ternary computation unit (TCU). In TCU, the expensive floating-point multipliers are replaced by multiplexers correspondingly as well as the size of SRAM equipped to CU can be reduced. However, the number of floating-point adders ($k \times k - 1$) has not changed.

C. Sparse-Ternary computation unit

Based on the basic architecture, we can easily implement the proposed STC by replacing the original CU module with Sparse-Ternary Computation Unit (STCU) module in its implementation. Figure 4 shows the proposed STCU architecture.

Comparing with the original CU, the features of STCU are:

- **Selector**: Figure 4 shows a comparison network designed for choosing the largest weights in $k \times k$ kernel and dropping others. Cause the kernel values are ternary, the comparators used in selectors are all for 1 bit ,which are area efficient. Actually, in our implementation we use a 9-input comparison network with 16 comparators to adapt the 3×3 kernel case. In order to keep high throughput, we pipeline one more stage in CU to fit this selector.
- **On-chip Storage**: Figure 5 shows the change of bit-width of kernels. The corresponding SRAM for storing STC kernel in each CU is shrunk 16 times (the same as TCU).

Table I shows the resource utilizations of different CUs described above.

V. Experimental results

We experimented our method with 2 datasets using Theano [1]: CIFAR-10 and SVHN to show how the proposed STC affects the CNNs training phase and test-time inference pass. The architectures of CNN for these 2 datasets are the same:

$$\begin{aligned}(2 \times 128C3) - MP2 - (2 \times 256C3) - MP2- \\ (2 \times 512C3) - MP2 - (2 \times 1024FC) - 10SVM\end{aligned} \tag{5}$$

where $C3$ is a 3×3 ReLU convolutional layer, MP is a 2×2 max-pooling layer, FC is a fully connected layer, and SVM is a SVM output layer. This architecture is inspired from VGGNet [4] and proved to be effective in BC [6]. The models are

TABLE I
Synthesis result of single CU using Vivado 2015, for kernel size 3*3.

CU Type	LUT	FF	BRAM	DSP
Original CU	3399	2370	4	9
Ternary CU	2811	2231	0.5	0
Sparse-Ternary CU	2255	1643	0.5	0

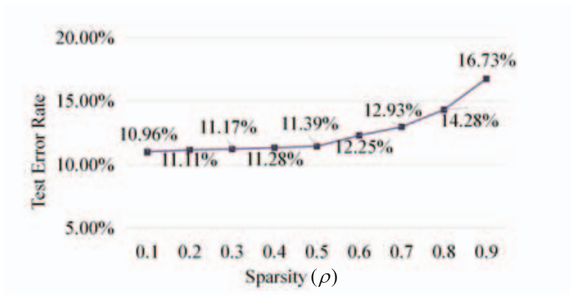


Fig. 6. STC model performance on CIFAR-10 using different sparsity parameters

TABLE II
TEST ERROR RATE COMPARISON. IN STC CASE, THE SPARSITY PARAMETER $\rho = 0.5$.

Method	CIFAR-10	SVHN
Full precision training + Full precision test	13.25%	2.79%
Binary training + Binary test	11.68%	2.47%
Ternary training + Ternary test	10.96%	2.42%
Sparse-Ternary training + Sparse-Ternary test	**11.39%**	**2.40%**

trained with SGD without momentum and batch normalization is used for accelerating learning. We do not use any data augumentation. For each dataset, all hyper-parameters are the same when different methods are tested.

A. Inference accuracy

As we have introduced in Section I, it's more meaningful to design the specialized architectures for performing inference using a trained network. Firstly, we tried different sparsity parameter in STC network trained on CIFAR-10 to show the effect of different sparsity on performance, the results are shown in Figure 6. STC actually becomes original TC when $\rho \leq 0.1$. At first, the performance degraded slightly until ρ reached 0.5. After that the increasing sparsity caused sharp degradation of performance. Therefore, $\rho = 0.5$ might be the best choice to do a reasonable tradeoff between performance and resources in hardware design.

When comparing the performance of various quantized-CNN, we used the deterministic form of sampling in both training and test phase. As shown in Table II, comparing with the original ternarized network, STC yielded a little higher error rate (11.39%) on CIFAR-10 and get a slightly lower error rate (2.41%) on SVHN. The result is all better than binarized network and full precision network.

To roughly estimate the sparsity in our model, we added up the number of ternary weights (-1, 0, 1) in the convolutional layers of original ternary model and 0.5 STC model. Table III shows the result. Taking the model trained on CIFAR-10 as a concrete example, there are 918331 zero-weights among 45745592 weights existing in convolutional layers. However, the implicit sparsity in original ternary network can not directly benefit the area-efficient hardware design. But as for 0.5 STC model, the sparsity is increased from 20.07% to 44.78% and there are at least 4 zero-weights in each 3×3 kernel.

TABLE III
DISTRIBUTION OF TERNARY WEIGHTS IN CONVOLUTIONAL LAYERS. IN STC CASE, THE SPARSITY PARAMETER $\rho = 0.5$.

Dataset	Model	0	-1	1	sparsity
CIFAR-10	Ternary Connect	918331	2054156	1602105	20.07%
	0.5 STC	2048484	1469523	1056585	44.78%
SVHN	Ternary Connect	892255	2118697	1563640	19.50%
	0.5 STC	2045457	1505711	1023424	44.71%

TABLE IV
TEST ERROR RATES ON MODELS TRAINED BY DIFFERENT METHODS AND TEST-TIME INFERENCE USING THE REAL-VALUE WEIGHTS. IN STC CASE, THE SPARSITY PARAMETER $\rho = 0.5$.

Training Method	CIFAR-10	SVHN
Full precision	13.25%	2.79%
Binary Connect [6]	11.56%	2.44%
Ternary Connect [9]	10.93%	2.38%
Spase Ternary Connect	**11.42%**	**2.44%**

B. Training performance

To test how the proposed STC affects the training performance, we used the stochastic form of sampling in training and real-value weights in test-time inference. Typically, we tested our method when setting sparsity parameter ρ to be 0.5, which means 4 weights in a 3×3 filter matrix are fixed to be zero in our model. Then we compared the result with full precision (single precision floating point), BC and TC.

Table IV shows the test error rates when we used different methods in training and used real-value weights during all test runs. There was negligible accuracy loss ($< 0.5\%$ on CIFAR-10, $< 0.1\%$ on SVHN) in STC training when comparing to original TC, but still achieved a better performance than full precison training and BC, which suggests that the proposed STC can also be used as a regularizer and its power is somewhere between original TC and BC.

Figure 7 shows the training behaviour of our models on CIFAR-10. This figure illustrates that STC slows down the training and gains a much better convergence than the conventional full precision training.

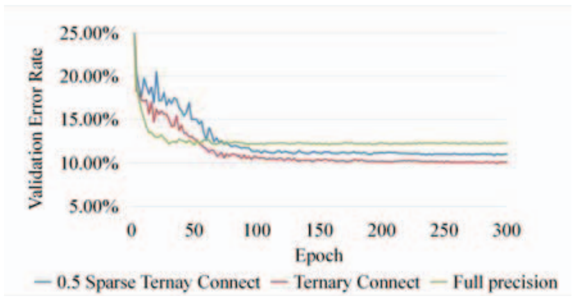


Fig. 7. Validation error rate at each epoch for 0.5 STC, TC and 32-bit single precision floating point training.

TABLE V
RESOURCE UTILIZATION OF 3^{rd} LAYER IN OUR MODEL, THE STC SPARSITY $\rho = 0.5$.

Impl.	LUT	FF	BRAM	DSP
Original CNN-MERP [11]	188887	138168	406	443
Ternary Impl.	160640	131496	46	11
Spase-Ternary Impl.	134263	103272	46	11

TABLE VI
PERFORMANCE COMPARISON WHEN FULLY USING THE RESOURCE ON BOARD.

Impl.	LUT	FF	BRAM	DSP	# of CUs	Throughput
Original	188887	138168	406	443	48	117Gop/s
Applying STC	192767	145339	89.5	8	72	176Gop/s
Device Cap.	203800	407600	445	840	-	-

C. FPGA implementation results

The proposed architecture and the base architecture are both implemented on Xilinx Kintex-7 xc7k325tffg900-2 platform using the Vivado 2015 tool. We implement the 3^{rd} convolutional layer with the 3×3 kernel size in our model. This layer is representative with enough input and output channels because 3×3 kernel size is widely used in several widely-known CNNs such as AlexNet [5], VGG [4] and ResNet [12]. The CUs in original implementation is replaced by corresponding TCUs or STCUs in TC and STCU implementation.

The implementation results are listed in Table V while keeping the frequency (137.0MHz) and throughput (117Gops/s) unchanged. Compared to the conventional implementation, the proposed STC implementation reduces almost 97.51% use of DSP and 88.67% of BRAM due to its feature inheriting from TC, and further reduces up to 28.92% LUT utilization and 25.26% FFs benefiting from its enhanced sparsity.

To the best of our knowledge, there are few works focused on binarized CNN accelerators especially on FPGA platform. It may be inappropriate to make evaluation between networks with different precision. Therefore we do a preliminary evaluation comparing with the original implementation. Since the resource utilization is significantly reduced by replacing original CUs with STCUs, the parallelism of STCUs in our architecture can be largely increased and then the performance improvement directly. By making a complete use of the resource on the same FPGA platform, the throughput increased up to 1.5$\times$ as shown in Table VI.

In our experiment, we simply evaluate the proposed method by altering the computation units. Certainly, more efficient architecture can be used to benefit from STC method and further improve the performance.

VI. CONCLUSION

In this work, we proposed Sparse Ternary Connect (STC) which is an improved method of Ternary Connect to reduce the complexity of computation in CNN models. In STC, enhanced sparsity is introduced in ternary convolution kernels to simplify the computation module in hardware. Then the corresponding hardware architecture for STC networks is proposed and implemented. The experimental results confirm that the proposed method can reduce considerable hardware resource utilization (by 28.9% of LUT, 25.3% of FF, 97.5% of DSP and 88.7% of BRAM) while retaining the accuracy superiority from the ternary training. From another point of view, 1.5$\times$ throughput can be achieved on the same platform. For future work, we will develop specific CNN accelerator to maximize the gain from the proposed method by balancing the utilization of resources and introduce more methods to simplify and optimize quantized CNNs.

ACKNOWLEDGEMENT

Thanks are due to members of Kimura-lab in Waseda University for their discussions. The work is supported in part by the fund from NEC.

REFERENCES

[1] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, et al., "Theano: A Python framework for fast computation of mathematical expressions", *arXiv preprint* arXiv:1605.02688, 2016.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems", *arXiv preprint* arXiv:1603.04467, 2016.

[3] R. Collobert, K. Kavukcuoglu and C. Farabet, "Torch7: A matlab-like environment for machine learning", in *BigLearn, NIPS Workshop*, No. EPFL-CONF-192376, Dec. 2011.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv preprint* arXiv:1409.1556, 2014.

[5] A. Krizhevsky, I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, pp.1097-1105, Dec. 2012.

[6] M. Courbariaux, Y. Bengio and J.P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations", in *Advances in neural information processing systems*, pp. 3123-3131, Dec. 2015.

[7] M. Courbariaux, I. Hubara, D. Soudry, R. EI-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.

[8] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks" , in *European Conference on Computer Vision*, pp.525-542, Oct. 2016.

[9] Z. Lin, M. Courbariaux, R.Memisevic and Y. Bengio, "Neural networks with few multiplications", *arXiv preprint* arXiv:1510.03009, 2015.

[10] R. Andri, L. Cavigelli, D. Rossi and L. Benini, "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights", in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pp. 236-241, Jul. 2016.

[11] X. Han, D. Zhou, S. Wang and S. Kimura, "CNN-MERP: An FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks", in *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pp. 320-327, Oct. 2016.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp.770-778, Jun. 2016.