Running Sparse and Low-Precision Neural Network: When Algorithm Meets Hardware

Bing Li†, Wei Wen†, Jiachen Mao†, Sicheng Li*, Yiran Chen†, Hai(Helen) Li†
†Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA
†{bing.li.ece, wei.wen, jiachen.mao, yiran.chen, hai.li}@duke.edu
* Hewlett Packard Labs, Palo Alto, CA, USA
*sicheng.li@hpe.com

Abstract— Deep Neural Networks (DNNs) are pervasively applied in many artificial intelligence (AI) applications. The high performance of DNNs comes at the cost of larger size and higher compute complexity. Recent studies show that DNNs have much redundancy, such as the zero-value parameters and excessive numerical precision. To reduce computing complexity, many redundancy reduction techniques have been proposed, including pruning and data quantization. In this paper, we demonstrate our co-optimization of the DNN algorithm and hardware which exploits the model redundancy to accelerate DNNs.

I. Introduction

DNNs made remarkable success in AI applications, particularly in computation visions. To achieve higher accuracy, DNN models tend to be larger and more complex. This results in challenges when deploying DNNs on real-world computing platforms [1]. At the same time, a lot of redundancies exist in the DNN models and appear as the zero-value parameters (including feature maps and weights), weight correlation and excessive data precision [2, 3]. The redundancies induce a considerable waste of the storage, memory bandwidth and computation resource. To provide improved computing effciency, various redundancy reduction approaches have been explored, from the sparse or pruning network, low-rank approximation (LRA), low data precision and data quantization.

In traditional sparsity methods, DNN training uses the regularization term to to avoid the overfitting and generate the sparse models [4, 5, 6]. Recent pruning methods remove the zero/low-value weights or activations to diminish the model size in the retraining process [7, 8, 9, 10]. However, these pruned models can gain the trivial speedup due to the non-structured connections. Thus, we propose a hardware-friendly regularization method to directly learn a structured compressed network and accelerate the DNNs on general hardware platforms [11, 12].

LRA can obtain a compact and approximate network model by matrix decomposition [13, 14, 15, 16]. However, to learn an accurate network structure, LRA needs the reiterations of decomposing, fine-tuning, etc., resulting in extra computation overhead. We capitalize on the weight redundancy in neural networks and propose a new method for LRA, which gains $2\times$ speedup on modern GPU without accuracy loss [17].

The training of large-scale DNN models requires the huge amount of input data and is often deployed on distributed systems, where data parallelism is adopted [18, 19, 20]. The gradient synchronization dominates the computation overhead in training process [21]. Since the arithmetic precision used on general platforms is redundant for DNN computation, the low precision gradient techniques have been widely studied to reduce gradient synchronization cost [21, 22, 23]. We explored the gradient quantization and proposed 3-level gradients to cut down the overhead of gradient synchronization and accelerate the distributed training [24].

In this paper, we walk through our proposed redundancy reduction schemes, including the software/hardware co-designs of the structured sparse neural network, an enhanced LRA algorithms and the ternary quantized gradients training for the distributed DNN [11, 12, 17, 24]. Firstly, we introduce the related works about redundancy reduction of DNN in Section II and then present the details of our designs in the aspects of sparse, LRA and low precision, respectively in Section III, IV, and V. Finally, Section VI concludes the paper.

II. Related Work

A. Sparsity regularization and pruning

The sparsification and pruning methods naturally reduce the demanded computation and storage resources and have been widely studied. Han et al. [7, 8, 10] removed the connected weights according to the pre-established threshold. The activations with zero input and output connections are also pruned. Therefore, the reduction of the network model size was significant and the pruned networks could fit in on-chip SRAM [8]. After pruning, the remain weights are stored in a random format, leading to irregular memory accesses in the computation that is difficult to achieve attractive acceleration without the dedicated hardware support [9].

Compared to the random sparsity and pruning methods, we focus on accelerating DNN on general platforms. We design a structured sparsity learning on the convolutional layers to generate a structured DNN model [11]. When deployed with the proposed model, the computation of DNNs on hardware platforms gains significantly speedup [12].

B. Low rank approximation

LRA decomposes a large model to a compact one with more lightweight layers by weight/tensor factorization, thus reduce

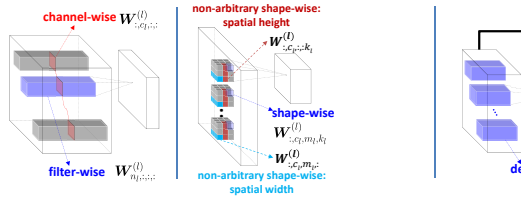

Fig. 1. The proposed *Structured Sparsity Learning* (SSL) for DNNs. [11, 12]

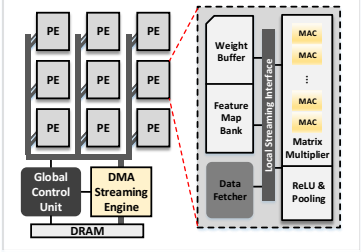

Fig. 2. The system architecture overview of the FPGA-based sparse CNN accelerator.

computation complexity. Denil et al. [13] studied different dictionaries to remove the redundancy between filters and channels in DNNs. Jaderberg et al. [15] explored filter and data reconstruction optimizations to attain optimal separable basis, and achieved 4.5× speedup on CPUs. Denton et al. [14] extended LRA to larger-scale DNNs, and achieved 2× speedup for the first two layers with 1% accuracy loss. Many new decomposition methods were proposed [16, 25] and effectively reduce the computation complexity and gains acceleration in state-of-the-art DNNs [26, 27]. However, the number of hyper-parameters in LRA method increases linearly with the layer depth, and the computation complexity increases linearly or even exponentially [15]. Therefore, we design a new LRA to train a DNN model with lower ranks and higher computation efficiency [17].

C. Low precision

State-of-the-art DNN designs have widely exploited reduced data precision to optimize the computing efficiency [28, 29]. Low-precision data representation can reduce the storage demand of the DNN models, lowering the data bandwidth requirements.

For distributed training in large DNN model, the gradients with smaller bit width can alleviate communication expense of gradient synchronization [22]. DoReFa-Net reduced the bit widths of weights, activations and gradients to 1, 2 and 6, respectively [30]. However, DoReFa-Net had 9.8% accuracy loss as it targeted at acceleration on single worker. F. Seide et al. applied 1-bit Stochastic Gradient Descent (SGD) to accelerate distributed training [22]. However, this method requires floating-point gradients to converge to a good initial point for the following 1-bit SGD. Our proposed *TernGrad* focuses on speeding up the distributed training by decreasing the communicated gradients to three numerical levels $\{-1; 0; 1\}$ [24].

III. Co-design in Sparse DNN

A. Methodology

We focus mainly on the Structured Sparsity Learning (SSL) on convolutional layers to regularize the structure of DNNs. Suppose the weights of convolutional layers in a DNN form a sequence of 4-D tensors $\boldsymbol{W}^{(l)} \in \mathbb{R}^{N_l \times C_l \times M_l \times K_l}$, where N_l, C_l, M_l and K_l are the dimensions of the l-*th* ($1 \leq l \leq L$) weight tensor along the axes of filter, channel, spatial height and spatial width, respectively. L denotes the number of convolutional layers. Then the proposed generic optimization target of a DNN with structured sparsity regularization can be formulated as:

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda R(\boldsymbol{W}) + \lambda_g \sum_{l=1}^{L} R_g\left(\boldsymbol{W}^{(l)}\right). \quad (1)$$

Here $\boldsymbol{W}$ represents the collection of all weights in the DNN; $E_D(\boldsymbol{W})$ is the loss on data; $R(\cdot)$ represents the original regularization method applying on every weight to avoid the over-fitting problem. $R_g(\cdot)$ is the additional structured sparsity regularization on each layer, using *Group Lasso*. The regularization of Group Lasso on a set of weights $\boldsymbol{w}$ can be represented as $R_g(\boldsymbol{w}) = \sum_{g=1}^{G} ||\boldsymbol{w}^{(g)}||_g$, where $\boldsymbol{w}^{(g)}$ is a group of partial weights in $\boldsymbol{w}$ and G is the total number of groups. The learned "structure" is decided by the way of splitting groups of $\boldsymbol{w}^{(g)}$.

We investigate and formulate the *filer-wise*, *channel-wise* and *shape-wise* structured sparsity in Fig.1. The optimization target of learning the *filer-wise* and *channel-wise Group Lasso* can be defined as

$$\lambda_n \sum_{l=1}^{L} \left(\sum_{n_l=1}^{N_l} ||\boldsymbol{W}^{(l)}_{n_l,:,:,:}||_g \right) + \lambda_c \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} ||\boldsymbol{W}^{(l)}_{:,c_l,:,:}||_g \right) \quad (2)$$

where $\boldsymbol{W}^{(l)}_{n_l,:,:,:}$ is the n_l-*th* filter and $\boldsymbol{W}^{(l)}_{:,c_l,:,:}$ is the c_l-*th* channel of all filters in the l-*th* layer. $\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}$ denotes the vector of all corresponding weights located at spatial position of (m_l, k_l) in the 2D filters across the c_l-*th* channel. The arbitrary shape-wise *Group Lasso* is formulated as:

$$\lambda_s \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} \sum_{m_l=1}^{M_l} \sum_{k_l=1}^{K_l} ||\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}||_g \right) \quad (3)$$

B. SSL based Hardware Designs

SSL based Mobile computing. We focus on the distributed mobile system with mobile computing and transmission characteristics and enhance the system performance by employing the *SSL* scheme.

We characterize the computing overhead of convolutional layers in DNN which is operated using GEneral Matrix to Matrix Multiplication (GEMM). GEMM first reshapes the 3D input feature map into 1D columns and then multiply the reshaped matrix with the filter matrix. The memory intensity of the reshaping steps and computing intensity of matrix-to-matrix multiplication steps are the critical causes of the high execution time in convolutional computations. We can train DNN to be less computing-intensive using filter-wise & channel-wise group lasso in Eq. 2 and less memory intensive with the help of the shape-wise group lasso in Eq. 3. Moreover, to reduce the communication cost among the work nodes, we use non-arbitrary group lasso regularization to zero out for all the filters in spatial height and spatial width in the shared parts of tensors. Thus, those tensors do not need to be transmitted, resulting in considerable decrease in transmission data size. The rationale of the non-arbitrary group lasso regularization is clarified in Fig. 1.

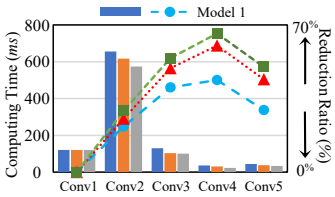

Fig. 3. Memory reduction (W_{mem}).[12]

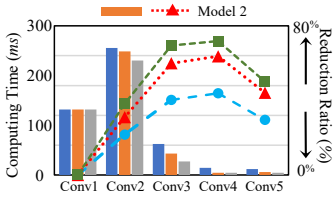

Fig. 4. FLOP reduction (W_{comp}). [12]

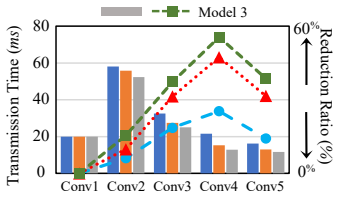

Fig. 5. Transmission reduction (W_{trans}). [12]

TABLE I
PERFORMANCE EVALUATION ON SPARSE CONV AND FC LAYERS OF ALEXNET ON IMAGENET.

Layer	Shape-wise Sparsity	Filter-wise Sparsity	Computation Efficiency CPU	GPU	VC707
Conv1	0%	9.4%	17.7%	11.5%	74.7%
Conv2	63.2%	12.9%	46.6%	24.9%	73.0%
Conv3	76.9%	40.6%	37.3%	9.3%	56.2%
Conv4	84.7%	46.9%	47.5%	13.6%	57.2%
Conv5	80.7%	0%	61.3%	9.1%	61.6%
Conv Total	**61.1%**	**21.9%**	**36.2%**	**13.7%**	**64.5%**
FC1	47.5%	9.7%	4.4%	1.1%	14.6%
FC2	43.5%	4.3%	4.7%	1.0%	13.1%
FC3	29.6%	3.3%	3.7%	0.6%	12.5%
FC Total	**40.2%**	**7.5%**	**4.3%**	**15.01%**	**13.4%**

SSL based FPGA Framework. To enable the DNN acceleration on FPGA, we propose a co-design framework to improve the efficiency of DNNs implementation.

Fig. 2 gives an overview of the proposed system architecture designed to implement sparse DNNs effectively. PEs array is the computation power of the accelerator. The number of PE is critical for the computing efficacy of DNNs but is limited by FPGA resources. *Data Fetcher* load vector arrays of an input feature map into *Feature Map Bank* at runtime. *Weight Buffer* insures the continuity of DMA service. The available hardware resource and memory bandwidth of the FPGA board include the number of MACs, the effective BRAM space and the memory bandwidth to off-chip DRAM. Assume r_{mac} units of resource are needed to construct a single MAC, R DSPs are available on chip and S_c MACs for the inner parallelism of a PE, $S_p \leq \frac{R}{r_{mac} \cdot S_c}$. Excluding the portion for interfacing to memory and other support functions, *Weight Buffer* and *Feature Map Bank* can use αM of BRAM. In terms of BRAM space, $S_p \leq \frac{\alpha M}{m_{sub} \cdot DW}$, where DW denotes the data width and m_{sub} is the size of sub-matrices. During computation, kernel weights and feature maps are streamed into PEs from the off-chip memory. T_{comp} should be larger than T_{IO}, otherwise implementing more PEs will incur low computation efficiency. According to the performance model, $S_p \leq \frac{N_{comp} \cdot B}{S_c \cdot N_{IO}}$, where B is the available memory bandwidth and N_{comp} and N_{IO} are respectively the number of computation data and data streamed from off-chip memory Therefore, the number of PEs in a particular FPGA platform can be formulated as: $S_p = \left\lfloor min(\frac{R}{r_{mac} \cdot S_c}, \frac{\alpha M}{m_{sub} \cdot DW}, \frac{N_{comp} \cdot B}{S_c \cdot N_{IO}}) \right\rfloor$. The experimental results of the proposed framework for CNN acceleration are presented in Section C.

C. Experiment

We present the effectiveness of SSL when optimizing the DNN for distributed mobile platform and the FPGA-based framework.

We evaluate SSL on the distributed mobile platform, which is established with five Google Nexus 5 smart phones. Three variants of *CaffeNet* models *Model 1*, *Model 2*, and *Model 3* are trained by *SSL*. Fig. 3 shows the 3 models can effectively reduce the memory usage as high as 65.9% in average. The reduced computing time is 27%, 35%, and 39% respectively. Fig. 4 depicts the practical performance on mobile devices of 3 sparsity regularization models and the results show significantly reduced FLOP are 33% – 57%. Contributed from the non-arbitrary shape-wise *Group Lasso*, the computing time of matrix multiplication steps reduces 25.7% on average, occupying 12% of total computing time. Given a distributed mobile network with 4 work nodes as shown in Fig. 5, the transmission time can be reduced by 22%.

Table I reports the performance of Conv1 $\sim$ Conv5 when the AlexNet is compressed according to shape-wise and filter-wise sparsity on FPGA-based platform. The evaluated platform is Xilinx FPGA VC707 board. Based on the performance and resource utilization model, we implement 32 PEs on the FPGA platform. The *computation efficiency* represents the ratio of the practical performance over the peak performance. Across all the sparse Conv layers, our accelerator achieves an average computation efficiency of 64.5% on VC707. Averagely, compared to the well-tuned CPU and GPU implementations, our FPGA implementation on VC707 improves computation efficiency 1.8× and 4.7×, respectively.

IV. FORCE REGULARIZATION BASED LRA

A. Methodology

There exists correlation among trained filters in DNNs and those filters lie in a low-rank space. Fig. 6 presents the results of correlation analysis of the first convolutional filters in *AlexNet* and *GoogLeNet* using *Linear Discriminant Analysis* (LDA). The results indicates high correlation among filters within a cluster. The filters are normalized to unit vectors and colored to four clusters by k-means clustering, and then projected to 2D space by LDA to maximize cluster separation. There are many lower-rank approximation methods. For the

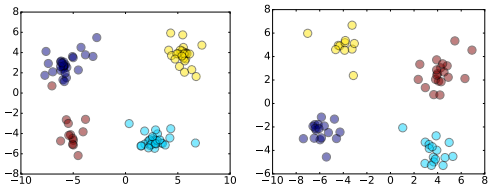

Fig. 6. *Linear Discriminant Analysis* (LDA) of filters in the first convolutional layer of *AlexNet* (left) and *GoogLeNet* (right). [17]

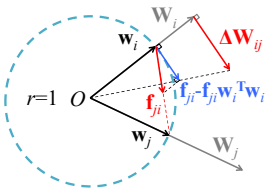

Fig. 7. *Force Regularization* to coordinate filters. [17]

cross-filter approximation, a convolutional layer, general a 4D tensor is approximated by a linear combination of the basis of a low-rank space. Since the linear combination essentially is a 1×1 convolution, a convolutional layer can be decomposed into two sequential lightweight convolutional layers. Thus, the computation complexity potentially decreases.

To achieve more accurate filter approximation in low rank, our basic rationale is nudging filters during the training such that the filters within a cluster are coordinated closer and some adjacent clusters are even merged into one cluster. We propose *Force Regularization* to coordinate more weight information into lower-rank space.

B. Force Regularization

Inspired by Newton's Laws, *Force Regularization* introduces extra gradients to data loss gradients, gently adjust the lengths and directions of data loss gradients so as to nudge filters to a more correlated state.

As illustrated in Fig. 7, suppose the filter $\mathcal{W}_n \in \mathcal{W}$ is reshaped as a vector $\mathbf{W}_n \in \mathbb{R}^{1\times CHW}$ and normalized as $\mathbf{w}_n \in \mathbb{R}^{1\times CHW}$ $(\forall n \in [1...N])$, with their origin at O. We introduce the pair-wise *attractive force* $\mathbf{f}_{ji} = f(\mathbf{w}_j - \mathbf{w}_i)$ $(\forall i, j \in [1...N])$ on $\mathbf{w}_i$ generated by $\mathbf{w}_j$. The gradient of *Force Regularization* to update filter $\mathbf{W}_i$ is defined as

$$\Delta\mathbf{W}_i = \sum_{j=1}^{N} \Delta\mathbf{W}_{ij} = ||\mathbf{W}_i|| \sum_{j=1}^{N} \left(\mathbf{f}_{ji} - \mathbf{f}_{ji}\mathbf{w}_i^T\mathbf{w}_i\right), \quad (4)$$

where $||\cdot||$ is the Euclidean norm. We derive the *Force Regularization* gradient from the *normalized* filters based on the following facts: (1) A normalized filter is on the unit hypersphere, and its orientation is the only free parameter we need to optimize; (2) The gradient of $\mathbf{W}_i$ can be easily scaled by the vector length $||\mathbf{W}_i||$ without changing the angular velocity. The regularization gradient in Eq. (4) is perpendicular to filter vector and can be efficiently computed by addition and multiplication.

The final updating of weights by gradient descent is

$$\mathbf{W}_i \leftarrow \mathbf{W}_i - \eta \cdot \left(\frac{\partial E(\mathcal{W})}{\partial \mathbf{W}_i} - \lambda_s \cdot \Delta\mathbf{W}_i\right)^{1}, \quad (5)$$

where $E(\mathcal{W})$ is data loss, η is learning rate and $\lambda_s > 0$ is the coefficient of *Force Regularization* to trade off the rank and accuracy. We select λ_s by cross-validation in this work.

Fig. 7 intuitively explains our method. Suppose each vector $\mathbf{w}_i$ is a rigid stick and there is a particle fixed at the endpoint. The particle has unit mass, and the stick is massless and can freely spin around the origin. Given the pair-wise attractive forces (*e.g.,* universal gravitation) $\mathbf{f}_{ji}$, Eq. (4) is the acceleration of particle i. As the forces are attractive, neighbor particles tend to spin around the origin to assemble together. If all filters could be extremely collapsed toward one point meanwhile maintain classification accuracy, it implies the filters are over-redundant and we can attain a very efficient DNN by decomposing it to a rank-one space.

In Eq. (4), $\mathbf{f}_{ji} = f(\mathbf{w}_j - \mathbf{w}_i)$ is the force function related to distance. We define ℓ_2*-norm Force* $f_{\ell_2}(\mathbf{w}_j - \mathbf{w}_i)$ as $\mathbf{w}_j - \mathbf{w}_i$ and name ℓ_1*-norm Force* $f_{\ell_1}(\mathbf{w}_j - \mathbf{w}_i)$ as $\frac{\mathbf{w}_j - \mathbf{w}_i}{||\mathbf{w}_j - \mathbf{w}_i||}$ in this work.

[1]The gradient regularization (*e.g.,* ℓ_2-norm) is omitted here for simplicity

TABLE II
THE SPEEDUPS OF *AlexNet* BY *Force Regularization*. [17]

Force	Top-1 error		conv3	conv4	conv5
None	43.21%	rank	184	201	146
ℓ_2-norm	43.25%	rank	124	106	129
None	43.21%	GPU	1.58×	1.21×	1.15×
ℓ_2-norm	43.25%	GPU	2.16×	2.03×	1.33×
None	43.21%	CPU	1.78×	1.60×	1.47×
ℓ_2-norm	43.25%	CPU	2.45×	2.76×	1.64×
None	43.21%	theoretical	1.79×	1.72×	1.63×

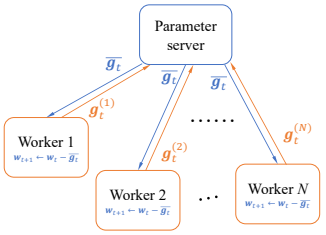

Fig. 8. Distributed SGD with data parallelism. [24]

C. Experiments

In our experiments, we first train DNNs with *Force Regularization*, then decompose DNNs using LRA methods and fine-tune them to recover accuracy. To prove the effective acceleration of *Force Regularization*, we adopt the speedup of state-of-the-art LRAs [13, 25] as our baseline. Our speedup is achieved in the case that the DNN filters are first coordinated by *Force Regularization* and then decomposed using the same LRAs. The practical GPU speed is profiled by the advanced hardware (NVIDIA GTX 1080) and software (cuDNN 5.0). The CPU speed is measured in Intel Xeon E5-2630 and ATLAS library.

Table II summarizes the speedups of PCA approximation of *AlexNet* with and without ℓ_2-norm *Force Regularization*. With ignorable accuracy difference, *Force Regularization* successfully coordinates filters to a lower-rank space and accelerates the testing by a higher factor, comparing with the state-of-the-art LRA. Similar results are observed when applying ℓ_1-norm force. In CPU mode of Table II, *Force Regularization* achieves $2\times$ speedup of total convolutional time. Table II also shows that practical speedup is different from theoretical speedup.

V. TernGrad in Distributed Deep Learning

A. Methodology

Fig. 8 formulates the distributed training problem of chronous SGD using data parallelism. At iteration t, a batch of training samples are split and fed into multiple ers ($i \in \{1, ..., N\}$). Worker i computes the gradients of parameters *w.r.t.* its input samples $\boldsymbol{z}_t^{(i)}$. All gradient first synchronized and averaged at *parameter server*, and sent back to update workers. Note that parameter serve most implementations [18, 20] are used to preserve shared *parameters*, while here we utilize it to maintain shared *gradients*. In Fig. 8, each worker keeps a copy of parameters locally. We name this technique as *parameter localization*. The parameter consistency among workers can be maintained by random initialization with an identical seed. *Parameter localization* changes the communication of parameters in floating-point form to the transfer of quantized gradients that require much lighter traffic.

B. TernGrad

Different from traditional distributed training, *TernGrad* will quantize all gradients $\boldsymbol{g}$ into ternary precision before sending to parameter server. Formally, with a random binary vector $\boldsymbol{b}_t$, $\boldsymbol{g}_t$ is ternarized as

$$\tilde{\boldsymbol{g}}_t = ternarize(\boldsymbol{g}_t) = s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t, \tag{6}$$

where $s_t \triangleq max\left(abs\left(\boldsymbol{g}_t\right)\right)$ is a *scaler* that can shrink ± 1 to a much smaller amplitude. $\circ$ is the Hadamard product. $sign(\cdot)$ and $abs(\cdot)$ respectively returns the sign and absolute value of each element. Compared with the default precision 32-bit gradients used in modern deep learning frameworks, *TernGrad* can at least reduce the *worker-to-server* traffic by a factor of $32/log_2(3) = 20.18\times$.

As aforementioned, *parameter localization* reduces *server-to-worker* traffic by pulling quantized gradients from servers. However, summing up ternary values in $\sum_i \tilde{\boldsymbol{g}}_t^{(i)}$ will produce more possible levels and thereby the final averaged gradient $\overline{\boldsymbol{g}_t}$ is no longer ternary. It emerges as a vital issue when workers use different scalers $s_t^{(i)}$. To minimize the number of levels, we propose a shared scaler $s_t = max(\{s_t^{(i)}\} : i = 1...N)$ across all the workers. We name this technique as *scaler sharing*. The sharing process has a small overhead of transferring $2N$ floating scalars. By integrating *parameter localization* and *scaler sharing*, the maximum number of levels in $\overline{\boldsymbol{g}_t}$ decreases to $2N + 1$. As a result, the *server-to-worker* communication reduces by a factor of $32/log_2(1 + 2N)$, unless $N \geq 2^{30}$.

To improve the convergence of *TernGrad*, we proposed *layer-wise ternarizing* and *gradient clipping*. As gradients are back propagated, the range of gradients in each layer changes. Instead of adopting a large global maximum scaler, we independently ternarize gradients in each layer using the layer-wise scalers. More specific, we separately ternarize the gradients of biases and weights by using Eq. (6), where $\boldsymbol{g}_t$ could be the gradients of biases or weights in each layer. To approach the standard bound more closely, we can split gradients to more buckets and ternarize each bucket independently. However, this will

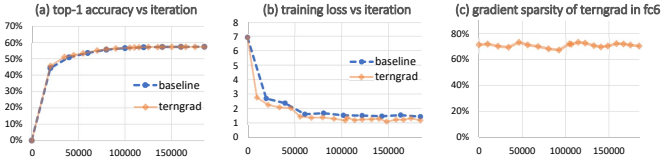

Fig. 9. *AlexNet* trained on 4 workers with mini-batch size 512. [24]

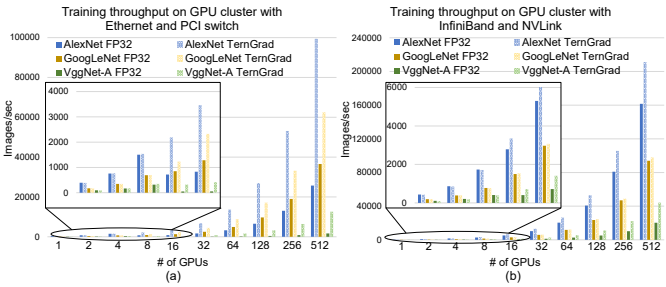

Fig. 10. Training throughput on two different GPUs clusters: (a) 128-node GPU cluster with 1Gbps Ethernet, each node has 4 NVIDIA GTX 1080 GPUs and one PCI switch; (b) 128-node GPU cluster with 100 Gbps InfiniBand network connections, each node has 4 NVIDIA Tesla P100 GPUs connected via NVLink. Mini-batch size per GPU of *AlexNet*, *GoogLeNet* and *VggNet-A* is 128, 64 and 32, respectively. [24]

introduce more floating scalers and increase communication. Layer-wise ternarizing can shrink the bound gap resulted from the dynamic ranges of the gradients across layers. However, the dynamic range within a layer still remains as a problem. We propose *gradient clipping*, which limits the magnitude of each gradient g_i in $\boldsymbol{g}$ as

$$f(g_i) = \begin{cases} g_i & |g_i| \leq c\sigma \\ sign(g_i) \cdot c\sigma & |g_i| > c\sigma \end{cases}, \tag{7}$$

where σ is the standard derivation of gradients in $\boldsymbol{g}$. c is a hyper-parameter to select, but we cross validate it only once and use the constant in all our experiments. In distributed training, gradient clipping is applied to every worker before ternarizing. By applying both *layer-wise ternarizing* and *gradient clipping* techniques, *TernGrad* can converge to the same accuracy as standard SGD. Removing any of the two techniques can result in accuracy degradation.

C. Experiments

We evaluate *TernGrad* by *AlexNet* and *GoogLeNet* trained on ImageNet and are performed by TensorFlow[31]. *TernGrad* converges to approximate accuracy levels regardless of mini-batch size. Fig. 9 plots training details vs. iteration when mini-batch size is 512. Fig. 9(a) shows that the convergence curve of *TernGrad* matches well with the baseline's, demonstrating the effectiveness of *TernGrad*. The training data loss in Fig. 9(b) shows that *TernGrad* converges to a slightly lower level, which further proves the capability of *TernGrad* to minimize the target function even with ternary gradients. A smaller dropout ratio in *TernGrad* can be another reason of the lower loss. Fig. 9(c) simply illustrate that on average 71.32% gradients of a fully-connected layer (fc6) are ternarized to zeros.

Fig. 10 presents the training throughput on two different GPUs clusters. Our results show that *TernGrad* effectively increases the training throughput for *AlexNet*,

VggNet-A and *GoogLeNet*. The speedup depends on the communication-to-computation ratio of the DNN, the number of GPUs, and the communication bandwidth. DNNs with larger communication-to-computation ratios (*e.g. AlexNet* and *VggNet-A*) can benefit more from *TernGrad* than those with smaller ratios (*e.g.*, *GoogLeNet*). Even on a very high-end HPC system with InfiniBand and NVLink, *TernGrad* is still able to double the training speed of *VggNet-A* on 128 nodes as shown in Fig. 10(b). Moreover, the *TernGrad* becomes more efficient when the bandwidth becomes smaller, such as 1Gbps Ethernet and PCI switch in Fig. 10(a) where *TernGrad* can have $3.04\times$ training speedup for *AlexNet* on 8 GPUs.

VI. Summary and Conclusions

In this work, we summarize our cross-layer optimizations to accelerate DNNs. *Structured Sparsity Learning* (SSL) regularizes the structures (*i.e.*, filters, channels, filter shapes, and layer depth) of DNNs and accelerate the DNNs on both the mobile distribute platforms and FPGA-based architecture. *Force Regularization* coordinates more weights into low-rank space, gains $2\times$ speedup for DNNs on GPU without accuracy loss. For the large-scale distributed learning, *TernGrad* uses three numerical levels $\{-1, 0, 1\}$ to represent gradients and significantly accelerates the DNN distributed learning.

Acknowledgements

This article was supported in part by NSF grants CNS-1717657, NSF SPX-1725456, NSF CCF-1744082 and DOE SC0017030. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DOE, or their contractors.

References

[1] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[2] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[4] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP)*, pages 1131–1135. IEEE, 2015.

[5] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, et al. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[6] Y-lan Boureau, Yann L Cun, et al. Sparse feature learning for deep belief networks. In *NIPS*, pages 1185–1192, 2008.

[7] Song Han, Jeff Pool, John Tran, and other. Learning both weights and connections for efficient neural networks. In *NIPS*, pages 1135–1143, 2015.

[8] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[9] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, pages 243–254, 2016.

[10] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, pages 1379–1387, 2016.

[11] Wei Wen, Chunpeng Wu, Yandan Wang, et al. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *NIPS*, pages 2074–2082. 2016.

[12] Jiachen Mao, Zhongda Yang, Wei Wen, et al. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *ICCAD*, pages 1–6. IEEE, 2017.

[13] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *NIPS*, pages 2148–2156, 2013.

[14] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, et al. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, pages 1269–1277, 2014.

[15] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[16] Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and othersS. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*, 2015.

[17] Wei Wen, Cong Xu, Chunpeng Wu, et al. Coordinating filters for faster deep neural networks. *arXiv preprint arXiv:1703.09746*, 2017.

[18] Jeffrey Dean, Greg Corrado, Rajat Monga, et al. Large scale distributed deep networks. In *NIPS*, pages 1223–1231. 2012.

[19] Mu Li, David G. Andersen, Jun Woo Park, et al. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2685095, 2014. USENIX Association.

[20] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

[21] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 440–445, 2017.

[22] Frank Seide, Hao Fu, Jasha Droppo, et al. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[23] Dan Alistarh, Jerry Li, Ryota Tomioka, et al. QSGD: randomized quantization for communication-optimal stochastic gradient descent. *CoRR*, abs/1610.02132, 2016.

[24] Wei Wen, Cong Xu, Feng Yan, et al. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *arXiv preprint arXiv:1705.07878*, 2017.

[25] Cheng Tai, Tong Xiao, Yi Zhang, et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[26] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, et al. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[27] Peisong Wang and Jian Cheng. Accelerating convolutional neural networks for mobile applications. In *Proceedings of the 2016 ACM on Multimedia Conference*, pages 541–545, New York, NY, USA, 2016. ACM.

[28] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[29] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[30] Shuchang Zhou, Yuxin Wu, Zekun Ni, et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[31] Martín Abadi, Ashish Agarwal, Paul Barham, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint:1603.04467*, 2016.