SparTANN: Sparse Training Accelerator for Neural Networks with Threshold-based Sparsification*

Hyeonuk Sim, Jooyeon Choi and Jongeun Lee
School of Electrical and Computer Engineering, UNIST, Ulsan, 44919 South Korea
Neural Processing Research Center, Seoul National University, Seoul, Korea

ABSTRACT

While sparsity has been exploited in many inference accelerators, not much work is done for training accelerators. Exploiting sparsity in training accelerators involves multiple issues, including where to find sparsity, how to exploit sparsity, and how to create more sparsity. In this paper we present a novel sparse training architecture that can exploit sparsity in gradient tensors in both back propagation and weight update computation. We also propose a single-pass sparsification algorithm, which is a hardware-friendly version of a recently proposed sparse training algorithm, that can create additional sparsity aggressively during training. Our experimental results using large networks such as AlexNet and GoogleNet demonstrate that our sparse training architecture can accelerate convolution layer training time by 4.20~8.88× over baseline dense training without accuracy loss, and further increase the training speed by 7.30~11.87× over the baseline with minimal accuracy loss.

1 INTRODUCTION

While sparsity has been amply explored to improve the efficiency of neural network hardware [5, 12, 13, 15], most of it is confined to inference hardware as opposed to training hardware. Neural networks must be first trained, which is to find the optimal set of synaptic weights for a given dataset, before making predictions for new data, which is called inference. Computationally, training includes the inference computation, plus finding the optimal weight adjustment, which happens to put greater demands on compute, memory, and arithmetic precision than inference. Hence, inference-only accelerators of neural networks appeared first [1, 2, 5, 12, 13, 15], and a much less number of training accelerators [3, 7, 8, 16, 19]. Training accelerators exploiting sparsity are even fewer.

In this paper we present a novel training accelerator architecture called SparTANN, which addresses three key problems of sparse training: where to find existing sparsity, how to exploit sparsity efficiently in hardware, and how to create even more sparsity. Though the problems have been partially addressed by previous work, no previous work addresses all three. SparTANN exploits sparsity in gradient tensors in both back propagation and weight update computation, which partly explains the superb efficiency of our accelerator compared with previous work [7]. Another ingredient is the efficient sparsity handling by SparTANN. On the algorithm level, it is similar to some sparse *inference* hardware such as [15], but details differ due to ours having to support training dataflow, which is definitely more complicated than that of inference.

Creating additional sparsity goes beyond simple zero skipping. Zero skipping doesn't alter the outcome of training, which is thus conservative, having neither quality degradation nor additional speed gain. On the other hand, sparse training algorithms [10, 11] can reduce the computational complexity of training by skipping even some non-zero elements without compromising the quality of training. The challenge is implementing them in hardware. Thus we propose a hardware-friendly algorithm, which is a single-pass algorithm that can be implemented with minimal hardware.

Our experimental results using large CNNs such as AlexNet and GoogleNet demonstrate that our sparse training architecture can accelerate convolution layer training by 4.20~8.88× over baseline dense training without accuracy loss, and further increase the training speed by 7.30~11.87× over the baseline with minimal accuracy loss (1.16%p or less).

*This work was supported by Samsung Advanced Institute of Technology, IITP grant funded by MSIT of Korea (No.2020-0-01336, Artificial Intelligence Graduate School Program (UNIST)), NRF grant funded by MSIT of Korea (No.2020R1A2C2015066), and Free Innovative Research Fund of UNIST (1.170067.01).
Jongeun Lee is the corresponding author (E-mail: jlee@unist.ac.kr).

ISLPED '20, August 10–12, 2020, Boston, MA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7053-0/20/08…$15.00
https://doi.org/10.1145/3370748.3406554

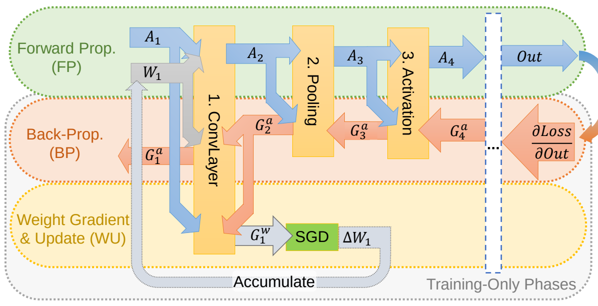


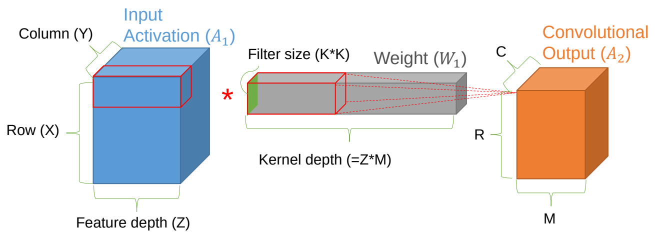
Figure 1: Computation flow of DNN training.

2 BACKGROUND AND RELATED WORK

2.1 Training Computation

Fig. 1 illustrates the computation flow of training, which can be divided into three *phases*. The first phase is *forward propagation* (FP), which is identical to inference, except that some intermediate results are also saved in the case of training (e.g., max index for max-pooling layer). FP applies the pre-defined operation (also known as the *forward* operation) of a layer such as convolution and ReLU to the input activation A_n of layer n, generating the output activation A_{n+1}, which becomes the input activation for the next layer.

The second phase is *back-propagation* (BP). Let G_{n+1}^a denote the loss gradient w.r.t. *output* activation of layer n. BP computes G_n^a from G_{n+1}^a, where the former, the loss gradient w.r.t. output activation of layer $n-1$, is also the loss gradient w.r.t. input activation of layer n. This is repeated for all layers except the first in the backward



```
for b in range(B):        # B is batch size
 for m in range(M):
  for r in range(R):
   for c in range(C):
    for z in range(Z):
     for $k_r$ in range(K):
      for $k_c$ in range(K):
       if FP:   $A^o_{\text{conv}}[b,m,r,c]$ +=
                $A^i_{\text{conv}}[b,z,r+k_r,c+k_c]$ * $W[m,z,k_r,k_c]$
       elif BP: $G^i_{\text{conv}}[b,z,r+k_r,c+k_c]$ +=
                $G^o_{\text{conv}}[b,m,r,c]$ * $W[m,z,k_r,k_c]$
       elif WU: $G^w_{\text{conv}}[m,z,k_r,k_c]$ +=
                $G^o_{\text{conv}}[b,m,r,c]$ * $A^i_{\text{conv}}[b,z,r+k_r,c+k_c]$
if WU: Update $W$ using $G^w_{\text{conv}}$ according to SGD rule
```

Figure 2: Definition of hyperparameters (top); Pseudo code showing the training computation of a convolution layer (bottom).

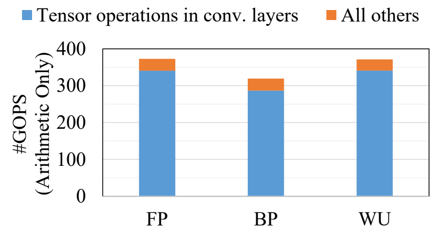


Figure 3: Computation complexity of each phase as measured by the number of arithmetic operations (Using all layers of the AlexNet, for 1 iteration; batch size is 256).

direction. For convolution and fully-connected layers, BP amounts to multiplying the weight transpose W_n^T to G^a_{n+1}; for element-wise layers, it can be done simply by multiplying the derivative.

The third phase, *weight gradient and update* (WU), is performed for all parametered layers. WU first computes the loss gradient w.r.t. weight, G^w_n, from G^a_{n+1} and A_n, and then updates the weight using a learning rule (e.g., SGD, Adam).

Sometimes it is convenient to refer to tensors as input/output. We use superscript i / o to denote the input / output *activation side*, respectively. Thus, for conv layer n, the following holds: $A_n = A^i_{\text{conv}}$ (input), $A_{n+1} = A^o_{\text{conv}}$ (output), $G^a_{n+1} = G^o_{\text{conv}}$ (input to BP/WU), $G^a_n = G^i_{\text{conv}}$ (output of BP), and $G^w_n = G^w_{\text{conv}}$ (output of WU).

2.2 Tensor Operations in Convolution Layers

Fig. 2 is an example pseudo code showing the computation of the three phases for a convolution layer. For brevity, stride and padding are assumed to be 1 and 0, respectively. As can be seen, at least for convolution layers, there is very high similarity among all three phases of computation, in terms of both computation complexity and computation pattern. On the network level, Fig. 3 shows the computation complexity of the three phases divided into two categories: tensor operations in convolution layers vs. all the other

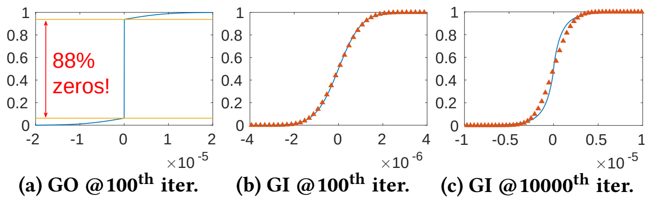


Figure 4: Cumulative distribution of gradient values in AlexNet Conv5 layer, which is followed by max-pooling. Red triangles show a normal distribution with the same mean and variance.

operations, with the latter including tensor operations in fully-connected layers as well as all element-wise operations. The result shows that the former dominates in training phases as well as in inference. For more recent CNNs that dispense with fully-connected layers, the portion of the former is even higher. Thus in this paper we focus on tensor operations in convolution layers.

2.3 Related Work

While no previous training hardware features *sparsification*, that is creating additional sparsity by skipping nonzero elements, it is shown on the algorithm level [10, 11] that sparsification in training can lead to significant reduction in computation complexity without compromising training quality, i.e., without degrading accuracy or increasing the number of training iterations. The algorithm is rather simple; it removes all but the top k elements in a gradient tensor, which is called top-k method. However the top-k method is inefficient for hardware implementation, since (i) it requires many comparison operations in the order of $O(n \log k)$, where n is the number of elements in the tensor, and (ii) it must traverse the gradient tensor twice, which can increase BP computation's latency and memory access significantly.

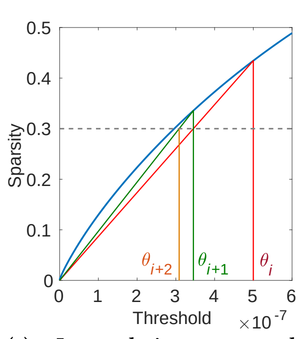
Previous sparse training hardware architectures [7, 19] do essentially zero skipping, removing only what is known to be ineffectual, but the mechanism to achieve it may vary. In particular, [7] exploits ineffectual computation in BP due to ReLU layer, i.e., negative operands need not be even created, which makes G^i_{conv}, the result tensor in convolution layers, about 50% sparse effectively. Contrarily we exploit sparsity in *operand* tensor, G^o_{conv}, which can be far sparser, enabling much higher speedup than the previous work (4∼10× vs. 2∼3× over dense) for certain CNNs. Also whereas the previous work's method is applicable to BP computation only, ours is equally effective to both BP and WU.

3 SPARSE TRAINING

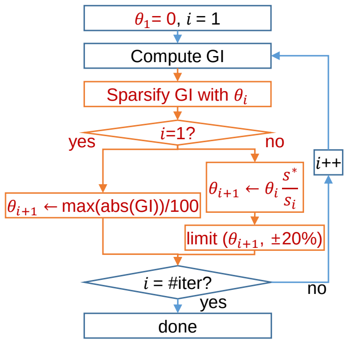
3.1 What is Sparse and What to Sparsify

Similar to inference, sparsity may be found in many tensors during training. For instance, the result of BP in ReLU, G^i_{ReLU}, is bound to have about 50% zeros. A max-pooling layer gives a far better deal; G^i_{MP} has only one-in-K^2 nonzero elements where $K \times K$ is the pooling size, assuming pooling windows don't overlap.

Consequently the sparsity in gradients varies greatly among layers. In the case of convolution layers of AlexNet, $G^i_{\text{conv}} \triangleq \text{GI}$, which is the result of BP, is hardly sparse whereas $G^o_{\text{conv}} \triangleq \text{GO}$, which is the result of BP in the following layer (max-pooling or ReLU), is very sparse as shown in Fig. 4. Since GO is input to both BP and WU (see Fig. 2), it seems very natural to develop a hardware architecture that utilizes zeros of GO (i.e., sparsity in operand tensor), which will make both BP and WU efficient.



(a) Interpolating expected threshold from GI of Fig. 4c



$\theta_1 = 0,\ i = 1$
Compute GI
Sparsify GI with θ_i
i=1?
yes
no
$\theta_{i+1} \leftarrow \max(\mathrm{abs(GI)})/100$
$\theta_{i+1} \leftarrow \theta_i \frac{s^*}{s_i}$
limit (θ_{i+1}, ±20%)
i++
i = #iter?
no
yes
done

(b) Flow chart of DTS (red colored)

Figure 5: Dynamic threshold scaling algorithm.

Now that GO is already very sparse, we can apply sparsification to other tensors. One good candidate is GI, which is not sparse even when GO is 88% sparse as shown in Fig. 4, but it has many near-zero values, which may be amenable to sparsification. Doing so would lower memory traffic and reduce effectual operations in subsequent layers, as well as possibly increasing the sparsity in the next G^o_{conv} (in another convolution layer).

However, since too much sparsification can backfire, defeating the purpose of sparsification, we must maintain the right sparsity level across all layers at all iterations, hopefully with as little hardware overhead as possible.

3.2 HW-friendly Sparsification Algorithm

The main lesson from the top-k sparse training algorithm [10] is that it is not absolute or relative value but ranking that matters most in determining whether to skip a nonzero element. For hardware implementation, we choose to use a threshold-based approach for its very low hardware overhead and little impact on latency. The threshold value, which we set per tensor per layer, is updated at runtime to ensure that the intended sparsity is achieved. To determine the right threshold, we use the statistics of a tensor in the previous iteration, based on the observation [6] that the distribution of a tensor's values changes gradually over time.

Our first attempt was to find the threshold value from a Gaussian function. If we assume that the mean and variance do not change significantly over one iteration, a Gaussian function can give the exact threshold for any target sparsity ratio. But there is one problem: data do not always follow Gaussian. Fig. 4 shows that the same tensor data can be perfectly Gaussian at one point, but deviate significantly (up to about 20%p in terms of sparsity in Fig. 8a) at a later point in time.

Thus we propose an iterative linear interpolation method, which does not require the Gaussian assumption. To illustrate the idea, let's say the target sparsity is $s^* = 0.3$, but the current threshold (θ_i = 5E-7) yields the sparsity of $s_i = 0.42$ (see Fig. 5a). Since CDF (cumulative distribution function) is likely symmetric, we use the CDF of absolute values, i.e., $f(|x|)$. We see that s_i is quite off from s^*, but learn that (θ_i, s_i) coordinate is on the CDF. Assuming the CDF doesn't change, and if the CDF can be linearized, we only need the coordinate of another point on the CDF to find the right threshold. Any coordinate will do, but one that is free is $(0, 0)$. By drawing a line crossing $(0, 0)$ and (θ_i, s_i), we can calculate the threshold for the next iteration θ_{i+1} as follows.

$$\theta_{i+1} \leftarrow \theta_i \cdot \frac{s^*}{s_i} \tag{1}$$

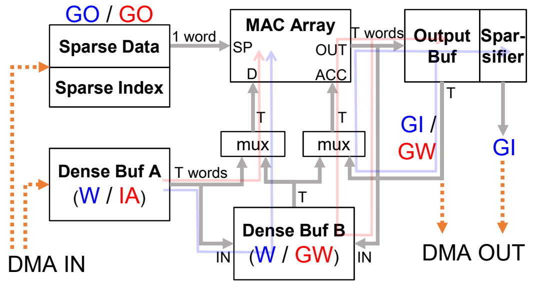


Figure 6: SparTANN architecture (for convolution layers). Note: IA $\triangleq A^i_{\text{conv}}$, GW $\triangleq G^w_{\text{conv}}$.

Interestingly, even if CDF is not linear, our method works well. This is because our prediction on the *direction* of threshold change is always correct. Also the fact that sparsity deviation ($s_i - s^*$), and thus threshold change ($\theta_{i+1} - \theta_i$), is usually very small, helps too. Our further analysis shows that our method under-compensates when $f'' < 0$ (where f is CDF), which is almost always the case. Our method is not only cheaper, but also more robust, than using two sparsity values from two previous iterations, because it is affected less by sampling noise. Also compared with the Gaussian distribution method, we only need to compute the sparsity value in each iteration, which is much cheaper than computing mean and variance.

Fig. 5b shows the flow chart of our algorithm, called *Dynamic Threshold Scaling (DTS)*. Since good training performance is important, we slowly increase threshold during the first iterations. (The iterations are training iterations; our algorithm is not iterative.) Initially we set $\theta_1 = 0$, which means no sparsification. At the second iteration we need an initial value, since (1) is useless when $\theta_i = 0$. Any small number would do, or an easy-to-compute formula is even better, e.g., $\max_j(|G^i(j)|)/100$. From the next iteration on, threshold is updated using (1). In order to prevent rapid change due to sampling noise or otherwise, we limit the threshold change per iteration to 20%. The hardware requirement of our DTS algorithm is a multiplier, a comparator, a divider, a counter and a few registers.

4 SPARSE TRAINING ARCHITECTURE

4.1 SparTANN Overview

Fig. 6 shows the block diagram of the SparTANN architecture for convolution layers. Inference typically requires much lower precision, which can be best done using a separate datapath. The diagram shows only the datapath for tensor operations, which account for the majority of computation in CNN training (see Fig. 3).

SparTANN does sparse-dense operations, meaning that in Fig. 2, W and A^i are treated as dense matrices while G^o is sparse. This decision is motivated by the fact that currently there is no known method of doing sparse-weight and/or sparse-activation training from scratch.

4.2 Sparse-Serial Dense-Parallel Architecture

A recurring problem in sparse handling architectures is low utilization of PEs (processing elements) [12], or alternatively, powerful but complex and large routing (i.e., non-computing) modules [5], both of which can significantly offset the efficiency of an architecture (colloquially called *sparsity tax*). We minimize sparsity tax, by employing sparse-dense operation and a dataflow where sparse input is consumed serially and dense input in parallel.

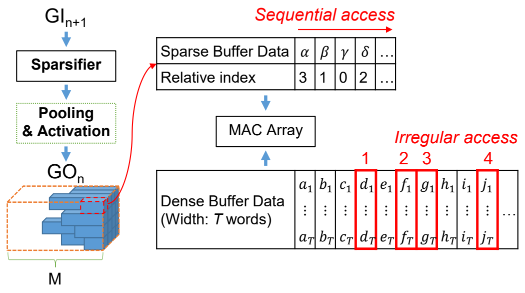


Figure 7: Sparse format and indexing example.

Our datapath, supporting both BP and WU, is centered around a MAC array. The MAC array has T multipliers and adders, where T is a design parameter, and performs scalar-vector multiplication and vector addition ($\alpha\vec{x} + \vec{y}$). In order to best utilize the MAC array, we parallelize BP and WU along the input channels (along the Z-loop in Fig. 2). Under this dataflow, each element of G^o is multiplied to Z elements of W (or A^i) in BP (or WU), respectively, meaning that G^o data are reused in Z number of MAC operations. Thus by storing W and A^i as T-dimensional vectors in *Dense Buf A* and G^o as scalar values in the sparse buffer (see Fig. 6), we can exploit the data reuse very efficiently, resulting in perfectly even utilization among PEs assuming T divides Z.

4.3 Sparse Format and Indexing Scheme

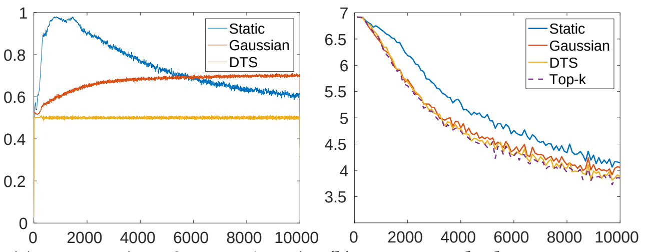
Specifically we use the CSC (compressed sparse column) format for G^o, where the *compressed column* corresponds to the output channel dimension (M-loop in Fig. 2). As illustrated in Fig. 7, at each cycle one element of G^o pops out, for which we need to provide the corresponding T-dimensional input vector of W for BP, or store the T-dimensional output vector in the corresponding location of G^w for WU. This requires irregular indexed access at the dense buffer in which W or G^w is stored, with the index coming from the sparse buffer. The sparse buffer stores relative index [2, 5, 15], which takes fewer bits than absolute index at the expense of one adder for conversion.

The sparse buffer is accessed sequentially but the dense buffer needs irregular access, with the index ranging up to M for both BP and WU. Thus we make *Dense Buf B* big enough to hold $M \times T$ words and use it to store input for BP and output for WU. Note that while sparsity implies that many of the M vectors are skipped, for different image pixels, different subsets of M vectors are accessed, resulting in some level of data reuse in the dense buffer. The sizes of other on-chip buffers can be determined based on how the loops in Fig. 2 are reordered, in a way very similar to designing inference accelerators [9].

Our sparsity handling architecture is very efficient, generating performance linearly proportional to sparsity with negligible overhead (see Section 5.2.1), meaning very low sparsity tax, which in turn renders our architecture to be useful for "dense" gradients as well, in contrast to other sparse architectures (e.g., [5, 12]).

5 EXPERIMENTS

To evaluate the efficacy of our approach, we show i) our DTS algorithm can track target sparsity very tightly, and ii) there is some sparsity level that gives significant computation reduction with little accuracy degradation (which was partially shown in previous work using top-k method), and also evaluate training efficiency, i.e., how well our architecture can reduce total training time.



(a) Sparsity (GI of Conv5 layer) **(b) Loss smoothed over 100 iter.**

Figure 8: Our DTS algorithm can track the target sparsity (50% in this example) very closely, giving almost the same training performance as top-k (AlexNet example).

Table 1: Test (or validation) accuracy (%) after training

	MNIST	CIFAR-10	AlexNet		GoogleNet	
			Top-1	Top-5	Top-1	Top-5
Ref. (Dense)	99.08	82.13	57.18	79.97	67.75	88.32
DTS-0.5	99.11	81.41	56.80	79.87	66.59	87.87
DTS-0.7	99.12	80.56	55.38	78.63	0.10	0.50
Random-0.5	99.06	81.00	55.78	78.89	48.87	73.76

5.1 Training Performance and Sparsity

5.1.1 Setup. We have implemented sparse training algorithms in C++, extending the Caffe framework [17]. For datasets and networks we use models from Caffe Model Zoo including MNIST CNN, CIFAR-10 CNN, AlexNet (using ImageNet), and GoogleNet (using ImageNet). We use the default training settings provided with the models; GoogleNet is trained using the `quick_solver` setting.

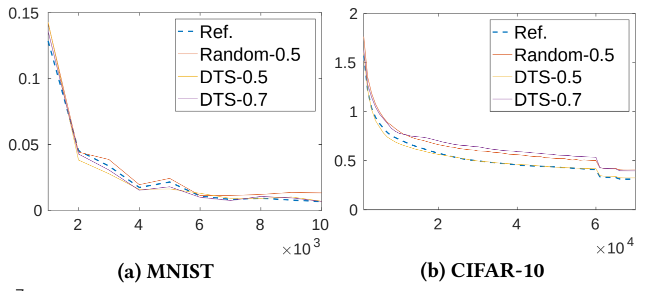
5.1.2 Tracking Sparsity Target. Fig. 8a compares the sparsity tracking accuracy of our algorithm vs. the Gaussian distribution method and static threshold. For static threshold, we use the median value obtained from the first iteration. Note that higher sparsity doesn't mean "better" here; it is about how tightly we can maintain the target sparsity, since both too much and too little sparsity can lower the training efficiency.

Interestingly the Gaussian distribution method[1] has a very large tracking error, which is mainly due to the fact that the underlying distribution is not always Gaussian. Using a static threshold can be much worse, and the widely varying curve suggests that the distribution of G^i can change over time quite dramatically. On the contrary, our method can track the target sparsity very tightly.

Tracking error translates into increased training loss in Fig. 8b, where we show the training loss of the top-k method as a reference. Our method gives an almost indistinguishable training curve as top-k throughout the entire training iterations, whereas the other methods show quite a gap.

5.1.3 GI Sparsity vs. Training Performance. Having established that our DTS algorithm can control the sparsity level very tightly, the next question is what the right sparsity should be. We compare two sparsity levels, 50% and 70% as referred to as DTS-0.5 and DTS-0.7, respectively, against a reference case, which is dense training. The training result is summarized in Table 1, which suggests that the optimal sparsity depends on the network. It may be hard to generalize but higher sparsity tends to work better with easier-to-train networks, which is quite understandable. In all cases DTS-0.5 shows good training performance, with the maximum accuracy loss of 1.16%p compared to dense training. We also compare the convergence speed in Fig. 9. Though it is not easy to read the graphs

[1]We set the threshold to $\mu + 0.67\sigma$ from the previous iteration's statistics, since for random variable X following $N(\mu, \sigma)$, $P(|X - \mu| \le 0.67\sigma) \cong 0.5$.



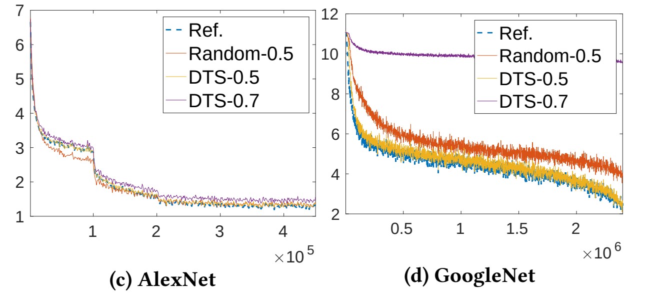


Figure 9: Our DTS-0.5 gives almost as good training performance as the reference case, i.e., dense training (x-axis: iterations, y-axis: loss smoothed over 1000 iterations).

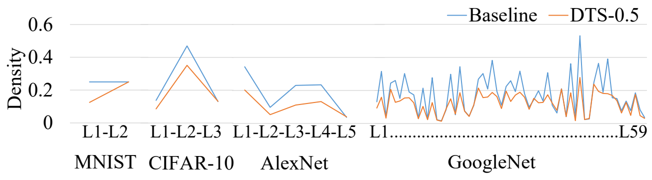


Figure 10: GO density with/without GI sparsification.

due to very high overlap, DTS-0.5 gives very similar training curves as dense training throughout the entire training iterations.

To test another aspect of DTS, we run random sparsification (Random-0.5), which is to sparsify stochastically 50% of G^i_{conv} irrespective of its value. *Random* works surprisingly well, better than DTS-0.7 in most cases, but again for difficult-to-train networks, it suffers a great deal of accuracy degradation. This result confirms that it is important to choose the right ones to drop in addition to enforcing the right sparsity level.

5.2 Training Efficiency Evaluation

5.2.1 Hardware Design and Implementation. We have implemented the proposed SparTANN architecture in Verilog and synthesized it using Synopsys Design Compiler with Samsung 65 nm standard cell library. Scratchpad buffers are modeled with Cacti [18]. DRAM power is estimated from access count with the unit access energy of 160 pJ/byte [5]. We set the design parameter T (= # MACs) to 32, targeting cost- and power-constrained devices. On-chip buffers have the following sizes: 4 KB sparse data buffer, 1 KB sparse index buffer, 68 KB dense (A+B) buffer, and 32 KB output buffer. In both software training and hardware implementation, we use single-precision floating-point arithmetic, which is necessary to guarantee convergence for large networks. The target frequency is set to 250 MHz. We conservatively estimate the overhead of the controller including an irregular address generator (i.e., adder) and interconnects to be 10% of the total on-chip area and power.

The total area including MAC array, on-chip buffers, and others such as sparsifier, control, and interconnects is estimated to be

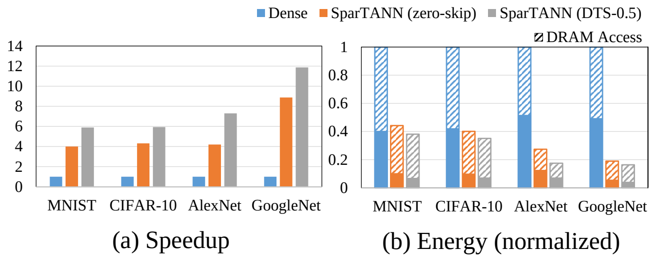


Figure 11: Efficiency of SparTANN in training (Convolution layers only).

4.74 mm^2, in which MAC array and on-chip buffers account for 0.69 mm^2 and 3.56 mm^2, respectively.

In terms of power, the steady-state power (after initial loading) of our SparTANN architecture is estimated to be about 0.20 W for BP and 0.29 W for WU (0.24 W on average). The higher power dissipation during WU is attributed to greater on-chip buffer accesses (especially Dense Buf B in Fig. 6) which is caused by lower data reuse. Data reuse during WU can be enhanced by utilizing a more specialized dataflow such as [3], which we did not consider to have a single datapath for both BP and WU. The large dense and output buffers consume about 64% of on-chip power in the worst case. On the other hand, sparsity-supporting hardware, i.e., sparsifier and sparse index buffer, has very low overhead in terms of both area and power (about 3%). Despite sparse tensors and on-chip buffers, DRAM power dissipation is still significant, more than half of the total power consumption. While it varies depending on the network and sparsity, DRAM access bandwidth and the total power dissipation are estimated to be about 1.95 GB/s and 0.58 W for AlexNet training, and 3.93 GB/s and 0.92 W for GoogleNet.

5.2.2 Efficiency of Sparse Training. We estimate the speedup and energy reduction of using our SparTANN architecture over dense training for convolution layers as follows. For dense training, we remove index buffer and sparsifier, which reduces area and power marginally. The latency of sparse training is computed from G^o_{conv} density, reflecting MAC under-utilization due to tiling (i.e., boundary tiles not fully utilizingg MACs). Computing the latency of dense training is more straightforward. Fig. 10 shows the G^o density (= 1 − sparsity) of the networks, confirming that most have less than 40% density and DTS-0.5 is effective in increasing sparsity of G^o in convolution layers.

Fig. 11 compares the speed and energy consumption of different architectures during BP and WU phases. For larger networks our SparTANN architecture can accelerate convolution layer training by 4.20~8.88× over the dense training via zero skipping, thanks to the large number of zeros generated by BP in post-convolution layers.[2] Our DTS method can further increase the training speed by 7.30~11.87× over the baseline with little accuracy loss, or 52% over zero skipping only. On-chip energy reduction follows a similar trend, reducing the energy consumption of training convolution layer by 78% (zero-skipping only) and 85% (with DTS) over the baseline, averaged for all the networks (geomean). DRAM energy also decreases significantly though not as much as on-chip energy reduction, because the amount of initial loading for dense tensors remains unchanged contrary to the proportional reduction of sparse

[2]Dense training hardware may employ a different, and potentially more efficient, architecture than what we assume for the dense training case, but a comparison with well-known dense hardware [16] reveals that our dense case has similar energy efficiency (GOPS/W) after technology scaling.

Table 2: Comparison with previous training accelerators

	Tech.	Freq.	Area	GOPS	Power	GOPS/W	Precision
Tegra X2	16	1300	n/a	1330	7.5	177	32-bit float
DaDianNao [16]	28	606	67.73	2090	15.97	131	32-bit fixed
TrainWare† [3]	65	100	n/a	32	0.05	617	16-bit
TNPU [8]	65	400	n/a	~130	n/a	n/a	32-bit float
DLAC* [19]	14	500	2.20	1390	n/a	n/a	32-bit float
Selective Grad.*† [7]	45	500	1.21	801	0.12	6867	32-bit fixed
SparTANN*	65	250	4.74	190	0.75	254	32-bit float
SparTANN* (T=64)	65	250	4.32	380	0.59	648	16-bit bfloat

Note: Omitted units are nm, MHz, mm^2, and W.

*Sparse architectures, for which GOPS number includes skipped operations.

†TrainWare supports WU only; Selective Grad. supports BP only.

tensors. The total energy reduction of larger networks is more substantial than smaller layers due to their higher speedup and less dominant DRAM energy. Normally, tensors of larger networks have higher chance of reuse, e.g. weights are reused for R×C times in inference (see Fig. 2). As a result, for large networks, total energy reduction is 83%~84% compared to dense training.

A training curve (Fig. 9d) may suggest that accuracy difference could be traded for training iterations. For GoogleNet, we calculate the average x-shift between DTS-0.5 and Ref. to be 40K iterations, or 1.67% of 2.4M iterations, which is much less than 34% speedup over zero-skipping only, achieved by DTS for GoogleNet.

Fine-tuning scenario: Our architecture can also be used for on-device training, in which case fine-tuning scenarios can be interesting as well. To see the applicability of our approach to such a training scenario, we ran an experiment. First we trained the AlexNet for CIFAR-10 (called *pretraining*), and then the pretrained model was fine-tuned for a more challenging dataset, CIFAR-100, for 40 epochs (called *retraining*). Our DTS-based sparse training was applied to retraining. Compared to the dense retraining, our architecture reduced the total retraining time by 4.17× with zero skipping and 4.84× with DTS-0.5 without any accuracy loss.

Light-weight CNN: To see the applicability of our sparse training to light-weight models, we profiled the training iterations of SqueezeNet-v1.1 [14], which shows similar recognition accuracy as AlexNet. We find that the network has about 70% zeros in G^o of convolutional layers, which means that SparTANN can achieve about 3.3× speedup over a dense architecture via simple zero skipping alone.

5.2.3 Comparison with Previous Training Accelerators. Table 2 compares various training hardware and a mobile GPU. First note that there is quite a variance among the compared architectures in terms of supported computation (BP, WU, or both) and the scope of hardware in reporting power (with or without on-chip buffers and DRAM). SparTANN supports both BP and WU, and its power figures include that of on-chip buffers and DRAM.

Compared with dense accelerators, such as Tegra X2 and DaDianNao, SparTANN shows higher GOPS/W though it uses an older technology. TrainWare, despite being dense hardware, shows very high energy efficiency. One reason is its use of 16-bit precision (unclear whether it is floating- or fixed-point) but it also optimizes on-chip memory access, lowering energy consumption significantly. For comparison, we implemented a 16-bit version of SparTANN using brain float [4], with the number of MACs doubled so that the dense buffer has the same width. The result is that SparTANN is more energy-efficient than TrainWare despite supporting both BP and WU on the same datapath. We note that the optimizations made by TrainWare and ours are largely orthogonal, and SparTANN's energy efficiency could be improved significantly by using separate, more specialized datapaths for BP and WU.

DLAC reports the highest area efficiency, but after scaling for technology and frequency differences, its area efficiency is much lower than that of ours (14 vs. 40 GOPS/mm^2). DLAC has relatively large overhead for sparse processing because PEs in DLAC include individual zero-skip logic. Compared with the other zero-skipping hardware, Selective Grad., our speedup is 7.30~11.87× over dense training, which is much higher than the reported speedup numbers by the previous work (2.63~3.13×) for similar-sized CNNs. This is because the previous work exploits sparsity from ReLU only (~ 50% sparsity) whereas SparTANN can exploit sparsity from max-pooling (> 80% sparsity) and ReLU as well as aggressive sparsification. While the previous work reports the highest energy efficiency, this figure doesn't include on-chip buffers, which typically dominate on-chip area and power, nor DRAM power. Since their baseline architecture [1] consumes about 62.38% of on-chip power in buffers and very high DRAM energy which is about 20× of on-chip energy including PEs and buffers, we expect their GOPS/W could decrease by at least 50×. Also it uses fixed-point arithmetic and a finer technology.

6 CONCLUSION

We presented sparse training hardware that can effectively hide sparse processing overhead, and a low-cost algorithm that can curb the impact of sparsification on training accuracy and convergence. Our detailed analysis of training performance as well as hardware efficiency suggests that sparse training can indeed offer a significant advantage over dense hardware for convolution layers.

REFERENCES

[1] Tianshi Chen *et al.* 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *ASPLOS 2014*. ACM.
[2] Yu-Hsin Chen *et al.* 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *ISCA 2016*. IEEE Press.
[3] Seungkyu Choi *et al.* 2018. TrainWare: A Memory Optimized Weight Update Architecture for On-Device Convolutional Neural Network Training. In *ISLPED 2018*. ACM.
[4] Google. 2019. *Using bfloat16 with tensorflow models.* https://cloud.google.com/tpu/docs/bfloat16
[5] Song Han *et al.* 2016. EIE: efficient inference engine on compressed deep neural network. In *ISCA 2016*. IEEE Press.
[6] Urs Köster *et al.* 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *NIPS 2017*. Curran Associates, Inc.
[7] Gunhee Lee *et al.* 2019. Acceleration of DNN Backward Propagation by Selective Computation of Gradients. In *DAC 2019*. ACM.
[8] Jiajun Li *et al.* 2019. TNPU: An Efficient Accelerator Architecture for Training Convolutional Neural Networks. In *ASPDAC 2019*. ACM.
[9] Atul Rahman et al. 2017. Design Space Exploration of FPGA Accelerators for Convolutional Neural Networks. In *DATE 2017*.
[10] X Sun *et al.* 2017. meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting. In *ICML 2017*. JMLR.org.
[11] X. Sun *et al.* 2020. Training Simplification and Model Simplification for Deep Learning : A Minimal Effort Back Propagation Method. *IEEE Transactions on Knowledge and Data Engineering* (2020).
[12] A. Parashar *et al.* 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ISCA 2017*.
[13] D. Kim *et al.* 2018. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Design Test* (2018).
[14] Forrest N. Iandola *et al.* 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).
[15] J. Albericio *et al.* 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *ISCA 2016*. IEEE Press.
[16] Y. Chen *et al.* 2014. DaDianNao: A Machine-Learning Supercomputer. In *MICRO 2014*.
[17] Yangqing Jia *et al.* 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR* abs/1408.5093 (2014).
[18] Shyamkumar Thoziyoor *et al.* 2008. CACTI 5.1. *HP Laboratories, April* (2008).
[19] G. Venkatesh *et al.* 2017. Accelerating Deep Convolutional Networks using low-precision and sparsity. In *ICASSP 2017*.