2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks

Youjie Li[1], Jongse Park[2], Mohammad Alian[1], Yifan Yuan[1], Zheng Qu[3], Peitian Pan[4], Ren Wang[5], Alexander Gerhard Schwing[1], Hadi Esmaeilzadeh[6], Nam Sung Kim[1]
[1]*University of Illinois, Urbana-Champaign*, [2]*Georgia Institute of Technology*, [3]*Tsinghua University*, [4]*Shanghai Jiao Tong University*, [5]*Intel Corporation*, [6]*University of California, San Diego*

***Abstract*—Training real-world Deep Neural Networks (DNNs) can take an eon (i.e., weeks or months) without leveraging distributed systems. Even distributed training takes inordinate time, of which a large fraction is spent in communicating weights and gradients over the network. State-of-the-art distributed training algorithms use a hierarchy of worker-aggregator nodes. The aggregators repeatedly receive gradient updates from their allocated group of the workers, and send back the updated weights. This paper sets out to reduce this significant communication cost by embedding data compression accelerators in the Network Interface Cards (NICs). To maximize the benefits of in-network acceleration, the proposed solution, named INCEPTIONN (In-Network Computing to Exchange and Process Training Information Of Neural Networks), uniquely combines hardware and algorithmic innovations by exploiting the following three observations. (1) Gradients are significantly more tolerant to precision loss than weights and as such lend themselves better to aggressive compression without the need for the complex mechanisms to avert any loss. (2) The existing training algorithms only communicate gradients in one leg of the communication, which reduces the opportunities for in-network acceleration of compression. (3) The aggregators can become a bottleneck with compression as they need to compress/decompress multiple streams from their allocated worker group.**

To this end, we first propose a lightweight and hardware-friendly lossy-compression algorithm for floating-point gradients, which exploits their unique value characteristics. This compression not only enables significantly reducing the gradient communication with practically no loss of accuracy, but also comes with low complexity for direct implementation as a hardware block in the NIC. To maximize the opportunities for compression and avoid the bottleneck at aggregators, we also propose an aggregator-free training algorithm that exchanges gradients in both legs of communication in the group, while the workers collectively perform the aggregation in a distributed manner. Without changing the mathematics of training, this algorithm leverages the associative property of the aggregation operator and enables our in-network accelerators to (1) apply compression for all communications, and (2) prevent the aggregator nodes from becoming bottlenecks. Our experiments demonstrate that INCEPTIONN reduces the communication time by 70.9$\sim$80.7% and offers 2.2$\sim$3.1$\times$ speedup over the conventional training system, while achieving the same level of accuracy.

I. Introduction

Distributed training [1–9] has been a major driver for the constant advances in Deep Neural Networks (DNNs) by significantly reducing the training time [7–9], which can take weeks [10] or even months [11]. Although distributing training unleashes more compute power, it comes with the cost of inter-node communications, proportional to the DNN size (e.g., AlexNet and ResNet-50 consist of 232 MB and 98 MB of weights). Moreover, accelerators, which further cut the computation time, make the cost of communication more pronounced [12, 13]. As illustrated in Fig. 1(a), state-of-the-art distributed training systems [6, 14, 15] are structured as a hierarchy of worker-aggregator nodes. In each iteration, the aggregator nodes gather the gradient updates from their subnodes, communicate the cumulative gradients upwards (see Fig. 1(a)) and send back the updated weights downwards. These gradients and weights of real-world DNNs are often hundreds of mega bytes (e.g., 525 MB for VGG-16 [16]), imposing significant communication loads on the network. This paper sets out to reduce this communication cost by embedding data compression accelerators in the Network Interface Cards (NICs).

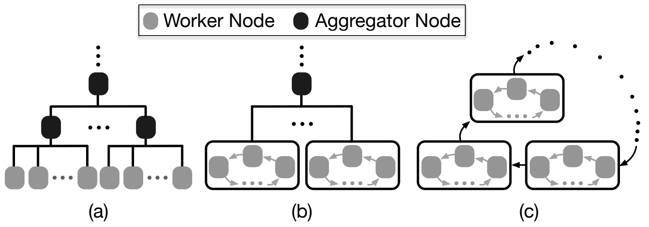


Fig. 1: (a) State-of-the-art hierarchical distributed training. (b) INCEPTIONN's distributed training algorithm in the conventional hierarchy. (c) Hierarchical use of INCEPTIONN's distributed algorithm.

Simply using general-purpose compression techniques and developing in-network accelerators for the compression would provide limited gains due to substantial hardware complexity and latency overhead. Thus, we instead propose a hardware-algorithm co-designed solution, dubbed INCEPTIONN[1], that offers a novel gradient compression technique, its in-network accelerator architecture, and a gradient-centric distributed training algorithm to maximize the benefits of the in-network acceleration. In designing and developing INCEPTIONN, we exploit the three following observations:

(1) Compared to weights, gradients are significantly more amenable to precision loss. Therefore, they lend themselves better to aggressive compression without requiring complicated mechanisms to alleviate their loss.
(2) The existing training algorithms communicate gradients in only one leg of the communication, which reduces the opportunities for compression and its in-network acceleration.

[1]**INCEPTIONN**: **I**n-**N**etwork **C**omputing to **E**xchange and **P**rocess **T**raining **I**nformation **O**f **N**eural **N**etworks

(3) Using compression can make the aggregators a bottleneck since they need to compress/decompress multiple streams of data corresponding to each of their subnodes.

Building upon these observations, INCEPTIONN first comes with a lightweight and hardware-friendly lossy-compression algorithm for floating-point gradient values. This compression exploits a unique value characteristic of gradients: their values mostly fall in the range between -1.0 and 1.0 and the distribution peaks tightly around zero with low variance. Given this observation, we focus on the compression of floating-point values in the range between -1.0 and 1.0 such that it minimizes the precision loss while offering high compression ratio. Moreover, the compression algorithm is developed while being conscious of the implementation complexity to enable direct hardware realization in the NIC. For seamless integration of the in-network accelerators with the existing networking software stack, we also provide a set of INCEPTIONN APIs that interface the accelerators with the traditional TCP/IP network stack and Open-MPI framework.

Although compressing the gradients is more effective than weights, its benefits cannot be fully utilized with conventional distributed training algorithms since they only pass the gradients in only one leg of the communication. Moreover, the aggregator would need to bear the burden of compressing/decompressing multiple streams. To tackle these challenges, INCEPTIONN comes with a gradient-centric, aggregator-free training algorithm, which leverages the following algorithmic insight to communicate gradients in both legs (see Fig. 1(b and c)). The aggregation operator (typically sum) is associative and thus, the gradients can be aggregated gradually by a group of workers. The intuition is to pass the partial aggregate from one worker to the other in a circular manner while they add their own contribution to the partial aggregate. This algorithm eliminates the need for a designated aggregator node in the group as it is conventional. This algorithm enables the distributed nodes to only communicate gradients and equally share the load of aggregation, which provides more opportunities for compressing gradients and improved load balance among the nodes. Fig. 1(b) visually illustrates the grouping view of our algorithm when it only replaces the leaf groups of conventional worker-aggregator hierarchy. Fig. 1(c) depicts the view when our algorithm replaces all the levels of hierarchy. These nodes form a worker group, which is the the building block of distributed training algorithms as depicted in all three organizations in Fig. 1.

The combination of (1) lossy compression algorithm for gradients, (2) NIC-integrated compression accelerator, and (3) gradient-centric aggregator-free training algorithm constructs a cross-stack solution, INCEPTIONN, that significantly alleviates the communication bottleneck without affecting the mathematics of DNN training. To demonstrate the efficacy of synergistically integrating the aforementioned three components, we train state-of-the-art DNN models such as AlexNet [17], VGG-16 [16], ResNet-50 [18]. Our experiments show that, INCEPTIONN reduces the communication time by 70.9∼80.7% and offers 2.2∼3.1× speedup in comparison with the conventional worker-aggregator based system, while achieving the same level of accuracy.

II. Distributed Training for Deep Neural Networks

A. Mathematics of Distributed Training

DNN training involves determining weights w of a predictor $\hat{y} = F(x, w)$, which processes input data x and yields a prediction $\hat{y}$. Supervised training finds w by minimizing a loss function $\ell(F(x,w),y^*)$, which compares the ground-truth output y^* with the prediction $\hat{y}=F(x,w)$ for given input data x and current w. Data and groundtruth are available in a training dataset $\mathscr{D}=\{(x,y^*)\}$ which is considered iteratively for many epochs. The commonly used optimization process for deep neural networks is *gradient descent*, which updates the weights in the opposite direction of the loss function's gradient, $g = \frac{\partial \ell_{\mathscr{D}}}{\partial w}$, where $\ell_{\mathscr{D}}$ denotes the loss accumulated across all samples in the set $\mathscr{D}$. Hence, the update rule that captures the *gradient descent* training is as follows:

$$w^{(t+1)} = w^{(t)} - \eta \cdot \frac{\partial \ell_{\mathscr{D}}}{\partial w^{(t)}} = w^{(t)} - \eta \cdot g^{(t)},$$

where $w^{(t+1)}$, $w^{(t)}$, and $g^{(t)}$ denote the next updated weights, the current weights, and the current gradient, respectively. The η parameter is the learning rate.

However, contemporary datasets $\mathscr{D}$ do not fit into the memory of a single computer or its GPUs, *e.g.* the size of popular datasets such as ImageNet [19] is larger than 200GB. To tackle this challenge, *stochastic gradient descent* emerged as a popular technique. Specifically, we randomly sample a subset from $\mathscr{D}$, often referred to as a minibatch $\mathscr{B}$. Instead of evaluating the gradient g on the entire dataset $\mathscr{D}$, we approximate g using the samples in a given $\mathscr{B}$, *i.e.*, we assume $g \approx \frac{\partial \ell_{\mathscr{B}}}{\partial w}$.

To parallelize this training process over a cluster, $\mathscr{D}$ can be divided into partial datasets $\mathscr{D}_i$ which are assigned to corresponding worker node i. Each worker can then draw minibatch $\mathscr{B}_i$ from its own $\mathscr{D}_i$ to calculate local gradient $g_i = \frac{\partial \ell_{\mathscr{B}_i}}{\partial w}$ and send g_i to an aggregator node to update the weights as follows:

$$w^{(t+1)} = w^{(t)} - \eta \cdot \sum_i \frac{\partial \ell_{\mathscr{B}_i^{(t)}}}{\partial w^{(t)}} = w^{(t)} - \eta \cdot \sum_i g_i^{(t)}.$$

The aggregator node, then, can send back the updated weights $w^{(t+1)}$ to all worker nodes. This mathematical formulation avoids moving the training data and only communicates the weights and gradients. Although the weights and gradients are much smaller than the training data, they are still a few hundreds of mega bytes and need to be communicated often.

B. Communication in Distributed Training

Building on this mathematical ground, there have been many research and development efforts in distributing DNN training [1–9]. State-of-the-art distributed training algorithms take the hierarchical worker-aggregator approach [6, 13–15], as illustrated in Fig. 2. In these algorithms, worker and aggregator nodes construct a tree where the leaves are the worker nodes that compute the local gradient ($g_i^{(t)}$) and the non-leaf nodes are the aggregator nodes that collect the calculated local gradients to update the weights ($w^{(t)}$) and send back the updated weights

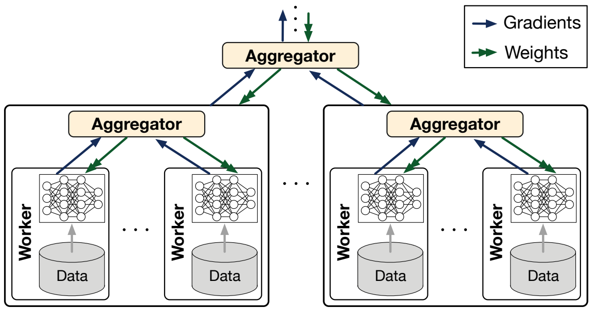


Fig. 2: Worker-aggregator approach for distributed training.

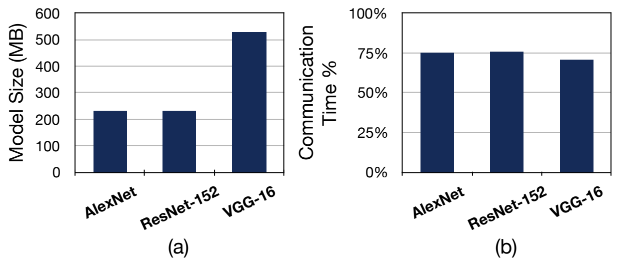


Fig. 3: (a) The size of weights (or gradients). (b) The percentage of the time spent to exchange g and w in total training time with a conventional worker-aggregator approach.

($w^{(t+1)}$) to worker nodes. The hierarchical reduction tree of aggregator nodes not only effectively disperses the networking and aggregation workload to distributed nodes, but also significantly reduces the size of system-wide data exchange by performing the intermediate aggregations. However, even with the hierarchical approach, each aggregator node should communicate with a group of worker nodes and aggregate the local gradients, which becomes the communication and computation bottleneck. Fig. 3 reports the exchanged weight/gradient size and the fraction of communication time when training state-of-the-art DNN models on a five-node cluster with 10Gb Ethernet connections. For instance, per each iteration, AlexNet requires 233 MB of data exchange for each of gradients and weights. Due to the large size of data exchange, 75% of training time for AlexNet goes to the communication. Some recent DNNs (*e.g.* ResNet-50: 98 MB) that have smaller sizes than AlexNet are also included in our evaluations (Sec. VIII). Nonetheless, as the complexity of tasks moves past simple object recognition, the DNNs are expected to grow in size and complexity [20]. The communication/computation ratio becomes even larger as the specialized accelerators deliver higher performance and reduces the computation time and/or more nodes are used for training.

III. Gradients for Compression

To reduce the communication overhead, INCEPTIONN aims to develop a compression accelerator in NICs. Utilizing conventional compression algorithms for acceleration is suboptimal since the complexity of algorithms will impose significant hardware cost and latency overhead. Thus, in designing the compression algorithm, we leverage the following algorithmic properties: (1) the gradients have significantly larger amenity to aggressive compression compared to weights, and (2) the gradients mostly fall in the range between -1.0 and 1.0 and the distribution peaks tightly around zero with low variance. These characteristics motivate the design of our lossy compression for gradients.

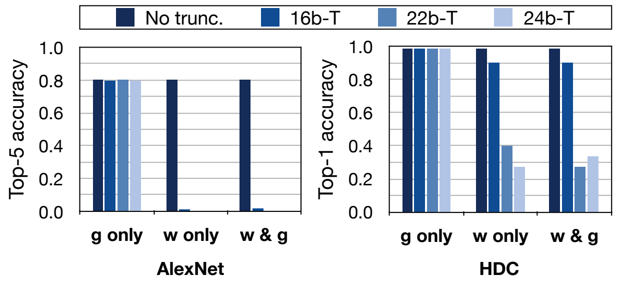


Fig. 4: Impact of floating-point truncation of weight w only, gradient g only, and both w and g on training accuracy of AlexNet and Handwritten Digit Classification (HDC). Floating-point truncation drops the LSB mantissa or even exponent bits of the 32-bit IEEE FP format. Xb-T represents truncation of X LSBs.

A. Robustness of Training to Loss in Gradients

Both weights (w) and gradients (g) in distributed training are normally 32-bit floating-point values, whereas they are 16 or 32-bit fixed-point values in the inference phase [21, 22]. It is widely known that floating-point values are not very much compressible with *lossless* compression algorithms [23]. For instance, using Google's state-of-the-art lossless compression algorithm, Snappy, not only offers a poor compression ratio of ∼1.5, but also increases the overall time spent for the training phase by a factor of 2 due to the computing overhead of compression. Thus, we employ a more aggressive *lossy* compression, exploiting tolerance of DNN training to imprecise values at the algorithm level. While lossy compression provides higher compression ratios and thus larger performance benefits than lossless compression, it will affect the prediction (or inference) accuracy of trained DNNs. To further investigate this, we perform an experiment using a simple lossy compression technique: truncating some Least Significant Bits (LSBs) of the g and w values. Fig. 4 shows the effect of the lossy compression on the prediction accuracy of both trained AlexNet and an handwritten digit classification (HDC) net. This result shows that the truncation of g affects the predictor accuracy significantly less than that of w, and the aggressive truncation of w detrimentally affects the accuracy for complex DNNs such as AlexNet. This phenomenon seems intuitive since the precision loss of w is accumulated over iterations while that of g is not.

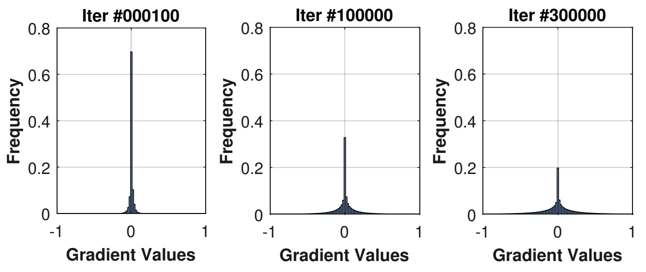
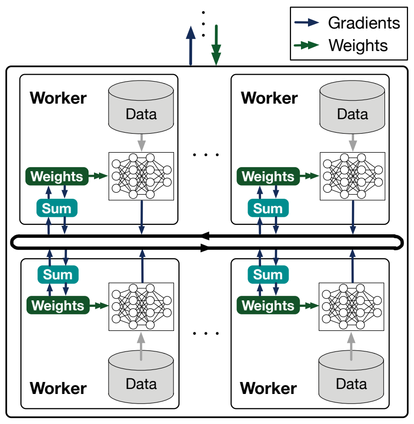
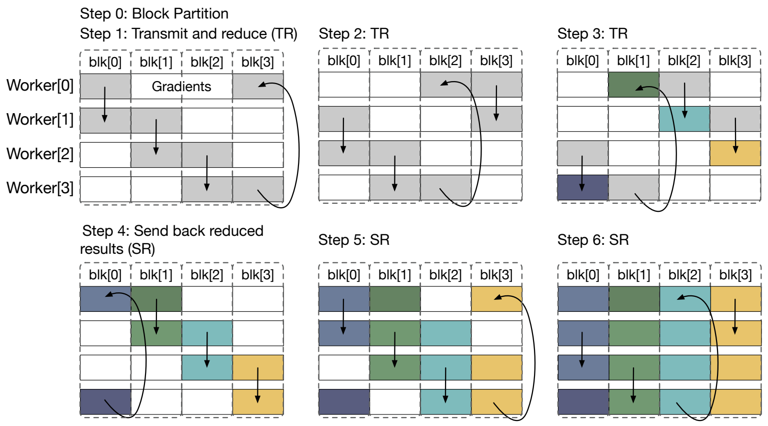


Fig. 5: Distribution of AlexNet gradient values at early, middle, and final training stages.



(a) Worker group organization.



(b) An example of distributed gradient exchange.

Fig. 6: INCEPTIONN gradient-centric distributed training algorithm in a worker group.

B. Tightness of Dynamic Range in Gradients

In designing the lossy compression algorithm, we leverage the inherent numerical characteristics of gradient values, *i.e.*, the values mostly fall in the range between -1.0 and 1.0 and the distribution peaks tightly around zero with low variance. We demonstrate the properties, analyzing the distribution of gradients at three different phases during the training of AlexNet. As plotted in Fig. 5, all the gradient values are between -1 and 1 throughout the three training phases and most values are close to 0. We also find a similar distribution for other DNN models. Given this observation, we focus on the compression of floating-point values in the range between -1.0 and 1.0 such that the algorithm minimizes the precision loss.

Our lossy compression algorithm (Section V) is built upon these two properties of gradients, and exclusively aims to deal with gradients. However, the gradients are only communicated in one direction in the conventional distributed training while the updated weights are passed around in the other direction. Therefore, before delving into the details of our compression technique and its hardware, we first discuss our training algorithm that communicates gradients in all the directions. Hence, this algorithm can maximize the benefits of INCEPTIONN's in-network acceleration of gradients.

IV. Gradient-Centric Distributed Training

Fig. 6(a) depicts the worker group organization of the INCEPTIONN training algorithm. In this algorithm, there is no designated aggregator node in the worker group. Instead, each worker node maintains its own model w (*i.e.*, model replica), and only exchanges and aggregates a subset of gradients g with two neighboring nodes after each iteration. Fig. 6(b) illustrates step-by-step the procedure of the algorithm using an example. At the beginning, every worker node starts with the same w_0 and INCEPTIONN evenly partitions gradient vectors into four blocks, blk[0], blk[1], blk[2], and blk[3] for four worker nodes. Every training iteration, each node loads and computes a mini-batch of data based on the current w and then generates a local g to be exchanged. Subsequently, INCEPTIONN exchanges and aggregates g in two phases.

```
On node[i] in a N-node cluster
1:  Initialize by the same model weights $w_0$, learning rate $\eta$
2:  for iteration $t = 0,...,(T-1)$ do
3:      Load a mini-batch $\mathscr{B}$ of training data
4:      Forward pass to compute current loss $\ell_{\mathscr{B}}$
5:      Backward pass to compute local gradient $g_i \leftarrow \frac{\partial \ell_{\mathscr{B}}}{\partial w}$
6:      (Compress local gradient $g_i \leftarrow Compress(g_i)$)
7:      // Gradient Exchange Begin
8:      Partition $g_i$ evenly into $N$ blocks
9:      for step $s = 1,...,N-1$ do
10:         Receive a block rb from node[(i−1)%N],
11:            then $\texttt{blk}_{\texttt{local}}[(\texttt{i}-\texttt{s})\%\texttt{N}] \leftarrow \texttt{rb} \bigoplus \texttt{blk}_{\texttt{local}}[(\texttt{i}-\texttt{s})\%\texttt{N}]$
12:         Send $\texttt{blk}_{\texttt{local}}[(\texttt{i}-\texttt{s}+1)\%\texttt{N}]$ to node[(i+1)%N]
13:     end for
14:     for step $s = N,...,2N-2$ do
15:         Receive a block rb from node[(i−1)%N],
16:            then $\texttt{blk}_{\texttt{local}}[(\texttt{i}-\texttt{s}+1)\%\texttt{N}] \leftarrow \texttt{rb}$
17:         Send $\texttt{blk}_{\texttt{local}}[(\texttt{i}-\texttt{s}+2)\%\texttt{N}]$ to node[(i+1)%N]
18:     end for
19:     // Gradient Exchange End
20:     (Decompress aggregated gradient $g_i \leftarrow Decompress(g_i)$)
21:     Update $w \leftarrow w - \eta \cdot g_i$
22: end for
```

Algorithm 1: INCEPTIONN gradient-centric distributed training algorithm for each worker node.

(P1) aggregation of gradients. worker[0] sends blk[0] to its next node, worker[1]. As soon as the blk[0] is received, worker[1] performs a sum-reduction on the received blk[0] and its own blk[0] (of worker[1]). This concurrently happens across all four workers ("Step 1"). This step is repeated two more times ("Step 2" – "Step 3") until worker[0], worker[1], worker[2], and worker[3] have fully aggregated blk[1], blk[2], blk[3], and blk[0] from all other 3 workers, respectively.

(P2) propagation of the aggregated gradients. worker[3] sends blk[0] to worker[0]. Now, worker[0] has blk[0] and blk[1]. This concurrently happens across all four workers and every worker has two fully aggregated blocks ("Step 4"). This step is repeated two more times ("Step 5" – "Step 6") until every worker has g which is fully aggregated from all four workers. Algorithm 1 formally describes the INCEPTIONN training algorithm to generalize it for an arbitrary number of workers, where the $\bigoplus$ denotes sum-reduction.

In summary, the INCEPTIONN training algorithm utilizes the

```
Input  : $f$: 32-bit single-precision FP value
Output : $v$: Compressed bit vector (32, 16, 8, or 0 bits)
         $t$: 2-bit tag indicating the compression mechanism
$s \leftarrow f[31]$       // sign
$e \leftarrow f[30:23]$    // exponent
$m \leftarrow f[22:0]$     // mantissa
if ($e \geq 127$) then
    $v \leftarrow f[31:0]$
    $t \leftarrow NO\_COMPRESS$   // 2'b11
else if ($e < error\_bound$) then
    $v \leftarrow \{\}$
    $t \leftarrow 0BIT\_COMPRESS$   // 2'b00
else if ($error\_bound \leq e < 127$) then
    $n\_shift \leftarrow 127 - e$
    $shifted\_m \leftarrow concat(1'b1, m) >> n\_shift$
    if ($e \geq error\_bound + \lceil (127 - error\_bound)/2 \rceil$) then
        $v \leftarrow concat(s, shifted\_m[22:8])$
        $t \leftarrow 16BIT\_COMPRESS$   // 2'b10
    else
        $v \leftarrow concat(s, shifted\_m[22:16])$
        $t \leftarrow 8BIT\_COMPRESS$   // 2'b01
    end
end
```

Algorithm 2: Lossy compression algorithm for single-precision floating-point gradients.

network bandwidth of every worker evenly unlike the worker-aggregator approach, creating the communication bottleneck. Furthermore, the algorithm performs the computation for aggregating gradients across workers in a decentralized manner, avoiding the computation bottleneck at a particular node. Lastly, the INCEPTIONN algorithm can be efficiently implemented with popular distributed computing algorithms such as Ring AllReduce [24].

V. Compressing Gradients

Compression. Algorithm 2 elaborates the procedure of compressing a 32-bit floating-point gradient value (f) into a compressed bit vector (v) and a 2-bit tag indicating the used compression mechanism (t). Note that this algorithm is described based on the standard IEEE 754 floating-point representation which splits a 32-bit value into 1 sign bit (s), 8 exponent bits (e), and 23 mantissa bits (m). Depending on the range where f falls in, the algorithm chooses one of the four different compression mechanisms. If f is larger than 1.0 (*i.e.*, $e \geq 127$), we do not compress it and keep the original 32 bits (NO_COMPRESS). If f is smaller than an error bound, we do not keep any bits from f (0BIT_COMPRESS). When the gradient values are in the range (error_bound $< f <$ 1.0), we should take a less aggressive approach since we need to preserve the precision. The simplest approach would be to truncate some LSB bits from the mantissa. However, this approach not only limits the maximum obtainable compression ratio since we need to keep at least 9 MSB bits for sign and exponent bits, but also affects the precision significantly as the number of truncated mantissa bits increases. Instead, our approach is to always set e to 127 and to not include the exponent bits in the compressed bit vector. Normalizing e to 127 is essentially multiplying $2^{(127-e)}$ to the input value; therefore, we need to remember the multiplicand so that it can be decompressed. To encode this information, we concatenate a 1-bit '1' at the MSB of m and shift it to the right by $127-e$ bits. Then we truncate some LSB bits from the shifted bit vector and keep either 8 or 16 MSB bits depending on the range of

```
Input  : $v$: Compressed bit vector (32, 16, 8, or 0 bits)
         $t$: 2-bit tag indicating the compression mechanism
Output : $f$: 32-bit single-precision FP value
if ($t = NO\_COMPRESS$) then
    $f \leftarrow v[31:0]$
else if ($t = 0BIT\_COMPRESS$) then
    $f \leftarrow 32'b0$
else
    if ($t = 8BIT\_COMPRESS$) then
        $s \leftarrow v[7]$
        $n\_shift \leftarrow first1\_loc\_from\_MSB\ (v[6:0])$
          $m \leftarrow concat(v[6:0] << n\_shift,\ 16'b0)$
    else if ($t = 16BIT\_COMPRESS$) then
        $s \leftarrow v[15]$
        $n\_shift \leftarrow first1\_loc\_from\_MSB\ (v[14:0])$
          $m \leftarrow concat(v[14:0] << n\_shift,\ 8'b0)$
    end
    $e \leftarrow 127 - n\_shift$
    $f \leftarrow concat(s, e, m)$
end
```

Algorithm 3: Decompression algorithm.

value. Consequently, the compression algorithm produces a compressed bit vector with the size of either 32, 16, 8, or 0 and 2-bit tag indicating the used compression mechanism.

Decompression. Algorithm 3 describes the decompression algorithm that takes a compressed bit vector v and a 2-bit tag t. When t is NO_COMPRESS or 0BIT_COMPRESS, the decompressed output is simply 32-bit v or zero, respectively. If t is 8BIT_COMPRESS or 16BIT_COMPRESS, we should reconstruct the 32-bit IEEE 754 floating-point value from v. First, we obtain the sign bit s by taking the first bit of v. Then we find the distance from MSB to the first "1" in v, which is the multiplicand used for setting the exponent to 127 during compression. Once we get the distance, e can be calculated by subtracting the distance from 127. The next step is to obtain m by shifting v to left by the distance and padding LSBs with zeros to fill the truncated bits during compression. Since we now have s, e, and m, we can concatenate them together as a 32-bit IEEE 754 floating-point value and return it as the decompression output.

VI. In-Network Acceleration of Gradient Compression

After applying compression algorithm in Section V, we may significantly reduce the amount of data exchanged among nodes in INCEPTIONN, but our final goal is to reduce the total training time. In fact, although researchers in the machine learning

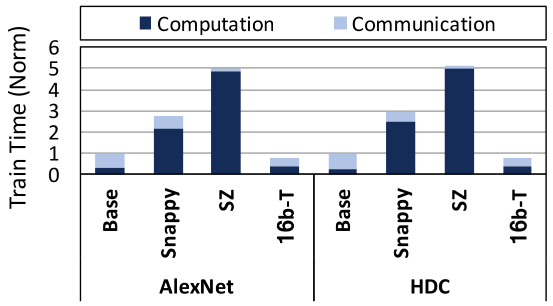


Fig. 7: Impact of software-based lossless (Snappy) and lossy (SZ) compression algorithms on the total training time of AlexNet and HDC. "Base" denotes baseline without compression. Xb-T represents truncation of X LSBs.

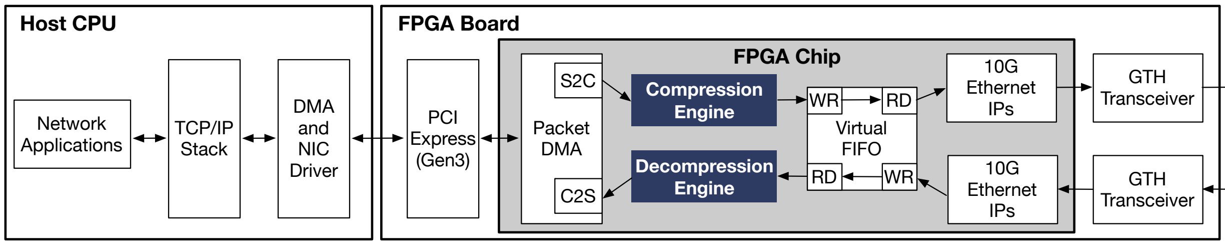


Fig. 8: Overview of NIC architecture integrated with compressor and decompressor.

community have proposed other compression algorithms [25–29], most of them did not report the total training wall-clock time after evaluating only the compression ratio and the impact of compression on training accuracy. Directly running these compression algorithms in software, though reducing the communication time, can place heavy burden on the computation resources and thus seriously increase computation time. Specifically, such compression algorithms need to run on the CPUs as GPUs cannot offer efficient bit manipulation (*e.g.*, packing some bits from floating-point numbers) compared to CPUs. Prior work [30] shows GPUs offer only ∼50% higher throughput at lower compression ratios than Snappy [31].

Fig. 7 shows that the training time increases by a factor of 2∼4× even when using the fastest lossless (Snappy) and lossy (SZ [32]) compression algorithms. Even a simple lossy truncation operation significantly increases the computation time, because simply packing/unpacking a large number of g values also significantly burdens the CPUs. This in turn considerably negates the benefit of reduced communication time as shown in Fig. 7, only slightly decreasing the total training time. Therefore, to reduce both communication and computation times, we need hardware-based compression for INCEPTIONN.

A. Accelerator Architecture and Integration with NIC

NIC architecture. To evaluate our system in a real world setting, we implement our accelerators on a Xilinx VC709 evaluation board [33] that offers 10Gbps network connectivity along with programmable logic. We insert the accelerators within the NIC reference design [34] that comes with the board. Fig. 8 illustrates this integration of the compression and decompression engines. For output traffic, as in the reference design, the packet DMA collects the network data from the host system through the PCIe link. These packets then go through the Compression Engine that stores the resulting compressed data in the virtual FIFOs that are used by the 10G Ethernet MACs. These MACs drive the Ethernet PHYs on the board and send or receive the data over the network. For input traffic, the Ethernet MACs store the received data from the PHYs in the virtual FIFOs. Once a complete packet is stored in the FIFOs, the Decompression Engine starts processing and passing it to the packet DMA for transfer to the CPU. Both engines use the standard 256-bit AXI-stream bus to interact with other modules.

Although hardware acceleration of the compression and decompression algorithms is straightforward, their integration within the NIC poses several challenges. These algorithms are devised to process streams of floating-point numbers, while the NIC deals with TCP/IP packets. Hence, the accelerators need to be customized to transparently process TCP/IP packets. Furthermore, the compression is lossy, the NIC needs to provide the abstraction that enables the software to activate/deactivate the lossy compression per packet basis. The following discusses the hardware integration and Section VI-B elaborates on the software abstraction.

Compression Engine. Not to interfere with the regular packets that should not be compressed, the Compression Engine first needs to identify which packets are intended for lossy compression. Then, it needs to extract their payload, compress it, and then reattach it to the packet. The Compression Engine processes packets in bursts of 256 bits, which is the number of bits an AXI interface can deliver in one cycle. Our engines process the packet in this burst granularity to avoid curtailing the processing bandwidth of the NIC. Our software API marks a packet compressible by setting the Type of Service (ToS) field [35] in the header to a special value. Since the ToS field is always loaded in the first burst, the Compression Engine performs the sequence matching at the first burst and identifies the compressible packets. If the ToS value does not match, compression is bypassed. The Compression Engine also does not compress the header and the compression starts as soon as the first burst of the payload arrives.

Fig. 9 depicts the architecture of the compression hardware. The payload burst feeds into the Compression Unit equipped with eight Compression Blocks (CBs), each of which performs the compression described in Algorithm 2. Each CB produces

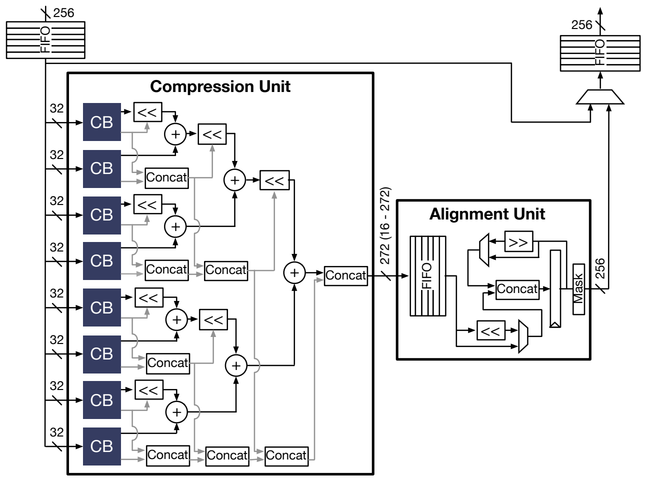


Fig. 9: 256-bit burst compressor architecture.

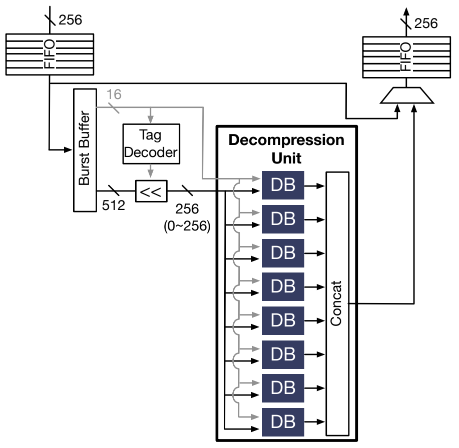


Fig. 10: 256-bit burst decompressor architecture.

a variable-size output in the size of either 32, 16, 8, or 0 bits, which need to be aligned as a single bit vector. We use a simple binary shifter tree that produces the aligned bit vector of which possible size is from 0 to 256. The 2-bit tags of the eight CBs are simply concatenated as a 16-bit vector. Finally, the aligned bit vector and tag bit vector are concatenated as the final output of the Compression Unit, of which size is at least 16 bits and can go up to 272 bits. For each burst, the Compression Unit produces a variable-size (16 – 272) bit vector; therefore, we need to align these bit vectors so that we can transfer the 256-bit burst via the AXI interface. The Alignment Unit accumulates a series of compressed bit vectors and outputs a burst when 256 bits are collected.

Decompression Engine. Similar to the Compression Engine, the Decompression Engine processes packets in the burst granularity and identifies whether or not the received packet is compressed through the sequence matching of the ToS field at the first burst. If the packet is identified as incompressible or the burst is header, decompression is bypassed. The payload bursts of compressible packets is fed into the decompression hardware, of which its architecture is delineated in Fig. 10. Since the compressed burst that contains 8 FP numbers can overlap two consecutive bursts at the Decompression Engine, reading only a single burst could be insufficient to proceed to the decompression. Therefore, the Decompression Engine has a Burst Buffer that maintains up to two bursts (i.e., 512 bits). When the Burst Buffer obtains two bursts, it feeds the 16-bit tag to the Tag Decoder to calculate the size of the eight compressed bit vectors. Given the sizes, the eight compressed bit vectors are obtained from the buffered 512 bits. Since each compressed bit vector has a variable size of either 32, 16, 8 or 0 bits, the possible size of the eight compressed bit vectors is from 0 and 256. These eight compressed bit vectors (0 – 256) and the tag bit vector (16) are fed into the eight Decompression Blocks (DBs) in the Decompression Unit, which executes the decompression algorithm described in Algorithm 3. Then, the Decompression Unit simply concatenates the outputs from the eight DBs and transfers it via the AXI interface. For the next cycle, Burst Buffer shifts away the consumed bits and reads the next burst if a burst (i.e., 256 bits) has been consumed and the left bits are fewer than a burst.

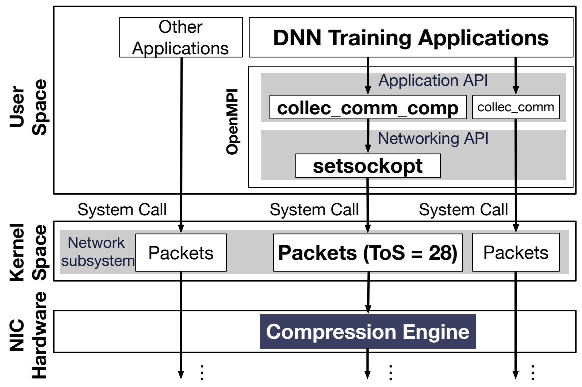


Fig. 11: Dataflow across the software stack and NIC hardware.

B. APIs for Lossy Compression of Gradients

As mentioned previously, we identify the context of a TCP/IP packet [36] by utilizing the ToS field in the IP header. ToS is an 8-bit field in the header of a TCP/IP packet and is used to prioritize different TCP/IP streams. We tag packets that need to be compressed/decompressed with a reserved ToS value of 0x28. For each socket connection, we can call the setsockopt function to set the ToS field or update it on the fly.

Fig. 11 demonstrates how we tag TCP/IP packets that need to be compressed/decompressed in the OpenMPI framework. It shows a scenario where we co-run DNN training application and some other networking applications on a server. To properly tag TCP/IP packets that require compression/decompression, we introduce MPI_collective_communication_comp, which is a specialized MPI_collective_communication API set. We implement our INCEPTIONN algorithm described in Section V without compression with the default MPI_collective_communication APIs. MPI_collective_communication_comp propagates a variable down to the OpenMPI networking APIs and sets the ToS option of the corresponding TCP sockets used for communication. We do not modify the Linux kernel network stack and the packets with ToS set to 0x28 reach to the NIC like regular TCP packets. Inside the NIC, a simple comparator checks the ToS field of each incoming packet; if the ToS field is set to 0x28, then the packet is sent to the compression engine, otherwise we do not perform compression for the outgoing packet. On a receiver node NIC, we have the same comparator for incoming packets. If the ToS field is set to 0x28, then we perform decompression on the packet. Otherwise, the received packet is a regular Ethernet packet and is directly sent to the processor for reception.

VII. Methodology

A. DNN Models

Table I enumerates the list of evaluated DNN models with the used hyper-parameters for training.

AlexNet. AlexNet [17] is a CNN model for image classification, which consists of 5 convolutional layers and 3 fully connected layers with rectified linear unit (ReLU) as the activation function. Before the first and the second fully connected layers, the dropout layers are applied. The model size of AlexNet is

TABLE I: Hyperparameters of different benchmarks.

Hyperparameter	AlexNet	HDC	ResNet-50	VGG-16
Per-node batch size	64	25	16	64
Learning rate (LR)	-0.01	-0.1	0.1	-0.01
LR reduction	10	5	10	10
Number of LR reduction iterations	100000	2000	200000	100000
Momentum	0.9	0.9	0.9	0.9
Weight decay	0.00005	0.00005	0.0001	0.00005
Number of training iterations	320000	10000	600000	370000

233 MB. For our experiments, we use 1,281,167 training and 50,000 test examples from the ImageNet dataset [19].

Handwritten Digit Classification (HDC). HDC [37–41] is a DNN model composed of five fully-connected layers, which performs Handwritten Digits Recognition. The dimension of each hidden layer is 500 and the model size is 2.5 MB. The used dataset is MNIST [42], which contains 60,000 training and 10,000 test images of digits.

ResNet-50. ResNet [18] is a state-of-the-art DNN model for the image classification task, which offers several variants that have different number of layers. Our experiments use the most popular variant, ResNet-50, which contains 49 convolution layers and 1 fully connected layer at the end of the network. ResNet-50 has a model size of 98 MB and uses the ImageNet dataset.

VGG-16. VGG-16 [16] is another CNN model for image classification, which consists of 13 convolutional layers and 3 fully connected layers. VGG-16 also uses ImageNet dataset and its model size is 525 MB.

B. Distributed DNN Training Framework

We develop a custom distributed training framework in C++ using NVIDIA CUDA 8.0 [43], Intel Math Kernel Library (MKL) 2018 [44], and OpenMPI 2.0 [45]. Note that INCEPTIONN can be implemented in publicly-released DNN training frameworks such as TensorFlow [46]. However, our custom distributed execution framework is more amenable for integration with software and hardware implementation of our lossy compression algorithm. In our custom training framework, all the computation steps of DNN training such as forward and backward propagations are performed on the GPU (also CPU compatible), while communication is handled via OpenMPI APIs. Besides, our framework implements diverse distributed training architectures and communication algorithms using various types of OpenMPI APIs to exchange gradients and weights.

C. Measurement Hardware Setup

We use a cluster of four nodes, each of which is equipped with a NVIDIA Titan XP GPU [47], an Intel Xeon CPU E5-2640 @2.6*GHz* [48], 32GB DDR4-2400T [49], and a Xilinx VC709 board [33] that implements a 10Gb Ethernet reference design along with our compression/decompression accelerators. We employ an additional node as an aggregator to support the conventional worker-aggregator based approach. We also extend our cluster up to eight nodes to evaluate the INCEPTIONN's scalability, while the rest of experiments are performed on the four-node cluster due to limited resources. All nodes are connected to a NETGEAR ProSafe 10Gb Ethernet switch [50]. Note that the state-of-the-art network architectures of datacenter at large Internet companies such as Google and Facebook use 1∼10Gbps network connections within a rack and 10∼100Gbps connections for the oversubscribed links between the top of rack switches [51, 52]. As the servers running the training applications are connected to the top of rack switches, we did not consider supporting 40∼100Gbps network connections for our experiments. Furthermore, we designed the compression/decompression accelerators such that they do not affect the operating frequency (100 MHz) and bandwidth while successfully demonstrating the full functionality with the modified NIC driver and OpenMPI APIs. Our distributed training framework runs concurrently on every node in our cluster and all performance evaluations are based on the real wall clock time. As we discover that the 10Gb Ethernet reference design implemented in a Xilinx VC709 board can achieve only ∼2.1 Gb due to inefficiency in its driver and design, we use Intel X540T1 10Gb Ethernet NICs [53] to measure the total training and communication times when we do not deploy hardware compression. That is, we use the Intel X540T1 NIC for all the baseline measurements. To measure the communication time after deploying hardware compression, we first measure the breakdown of communication time (*e.g.*, driver time, NIC hardware time, and TX/RX time through links) from both NICs based on Xilinx VC709 board and Intel X540T1 10Gb Ethernet NICs. Then, we scale the TX/RX time through the link of the Intel NIC based on a compression ratio corresponding to a given iteration to calculate the total communication time while accounting for the compression/decompression time.

TABLE II: Detailed time breakdown of training different benchmarks using the worker-aggregator based five-node cluster. Measurements are based on 100-iteration training time in seconds.

Steps	AlexNet		HDC		ResNet-50		VGG-16	
	Abs.	Norm.	Abs.	Norm.	Abs.	Norm.	Abs.	Norm.
Forward pass	3.13	1.6%	0.08	4.9%	2.63	3.5%	32.25	4.3%
Backward pass	16.22	8.3%	0.07	4.3%	4.87	6.5%	142.34	17.3%
GPU copy	5.68	2.9%	-	-	2.24	3.0%	12.09	1.5%
Gradient sum	8.94	4.6%	0.09	5.2%	3.68	4.9%	19.89	2.4%
Communicate	148.71	75.7%	1.36	80.2%	60.58	80.2%	583.58	70.9%
Update	13.67	7.0%	0.09	5.3%	1.55	2.1%	30.50	3.7%
Total training time	196.35	100.0%	1.7	100.0%	75.55	100.0%	823.65	100.0%

VIII. Evaluation

A. Performance Improvement with INCEPTIONN

We implement the conventional worker-aggregator training algorithm in a cluster of four workers and one aggregator, as the reference design. Table II shows a detailed breakdown of the training time of AlexNet, HDC, ResNet-50 and VGG-16, on the cluster. We report both the absolute and normalized time for 100 iterations of training. Irrespective of which DNN model we consider, Table II shows that (1) less than 30% of the wall-clock time is spent for local computations including the forward/backward propagations and update steps, and (2) more than 70% of the time is used for communication, which clearly indicates that the communication is the bottleneck.

Fig. 12 first compares the training time of the reference design (**WA**) with that of the INCEPTIONN (**INC**), when both run for the same number of iterations/epochs without applying compression. We also provide the training time

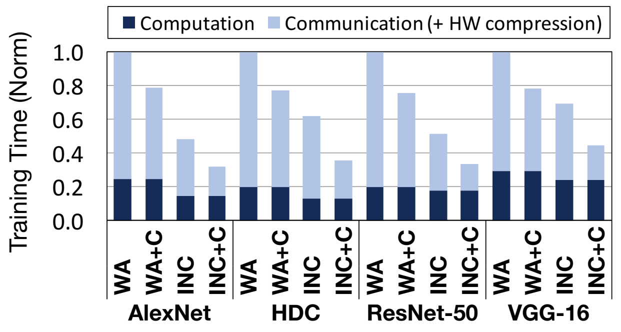


Fig. 12: Comparison of training time between the worker-aggregator based approach (WAx) and the INCEPTIONN (INCx) with and without hardware-based compression in NICs. WA denotes the worker-aggregator baseline without compression, and WA+C denotes WA integrated with our compression only on gradient communication with an error bound of 2^{-10}. INC denotes INCEPTIONN baseline without compression, and INC+C denotes with our compression given an error bound of 2^{-10}. Training time is measured in a cluster of four workers for INCx and one more aggregator for WAx. Note that these measurements are based on the same number of training iterations.

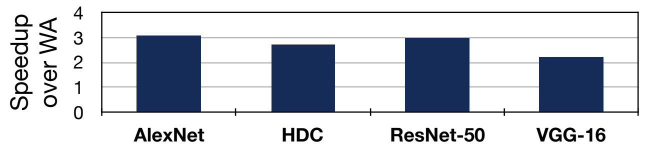
breakdown between computation and communication. This result shows that even in a small cluster without compression, the INCEPTIONN's training algorithm offers 52%, 38%, 49%, and 31% shorter total training time than the worker-aggregator based algorithm for AlexNet, HDC, ResNet-50 and VGG-16, respectively. This is due to 55%, 39%, 58%, and 36% reduction in communication time in comparison with the reference design.

Intuitively, INCEPTIONN is much more communication-efficient, because it not only removes the bottleneck link, but also enables concurrent utilization of all the links among nodes. Besides, this balanced gradient exchange also contributes to the reduction of computation time as the gradient summation is done by all the nodes in a distributed manner, whereas the worker-aggregator based algorithm burdens the designated aggregator nodes to perform the aggregation of the gradients collected from a group of subnodes.

Furthermore, Fig. 12 compares the training time of the reference design and INCEPTIONN system, when both are equipped with our gradient compression (**WA+C, INC+C**). From the result, we see that the conventional worker-aggregator based approach can still benefit from our compression with a $\sim$30.8% reduction in communication time compared to its baseline (**WA**), although only one direction of communication is applicable for compression. On the other hand, our gradient-centric INCEPTIONN algorithm offers maximized compression opportunities such that INCEPTIONN with hardware compression (**INC+C**) gives $\sim$80.7% and $\sim$53.9% lower communication time than the conventional worker-aggregator baseline (**WA**) and INCEPTIONN baseline (**INC**), respectively. Therefore, the full INCEPTIONN system (**INC+C**) demonstrates a training time speedup of 2.2$\sim$3.1$\times$ over the conventional approach (**WA**) for the four models trained over the same number of epochs.

B. *Effect of INCEPTIONN Compression on Final Accuracy*

The accuracy loss in gradients due to lossy compression may affect the final accuracy and/or prolong the training because of the necessity to run more epochs to converge to the



	AlexNet		HDC		ResNet-50		VGG-16	
	WA	INC+C	WA	INC+C	WA	INC+C	WA	INC+C
Training Time	175h	56h	170s	64s	378h	127h	847h	384h
# of Epochs	64	65	17	18	90	92	74	75
Final Accuracy	57.2%	57.2%	98.5%	98.5%	75.3%	75.3%	71.5%	71.5%

Fig. 13: Speedup of INCEPTIONN over the conventional approach when both achieve the same level of accuracy. We use the same notations with Fig. 12.

lossless baseline accuracy. To understand the effect of our lossy compression on accuracy (and on prolonged training), we take the conventional worker-aggregator system (**WA**) and the INCEPTIONN system (**INC+C**), and train the models until both systems converge to the same level of accuracy. Fig. 13 presents the total number of epochs and the final speedup of INCEPTIONN system over the conventional training system to achieve the same level of accuracy. From this, we observe that only a modest number of more epochs (1 or 2) are required to reach the final accuracy and thus INCEPTIONN system still offers a speedup of 2.2$\times$ (VGG-16) to 3.1$\times$ (AlexNet) over the convention approach, which matches the performance in Sec. VIII-A. Furthermore, we find that the extra number of training epochs is small but essential, without which an accuracy drop of $\sim$1.5% might incur.

C. *Evaluation of INCEPTIONN Compression Algorithm*

Fig. 14 compares the compression ratios among various lossy compression schemes, and the impact of these compressions on relative prediction accuracy of DNNs which are trained through our training algorithm for the same number of epochs. Specifically, we evaluate truncation of 16, 22, and 24 LSBs

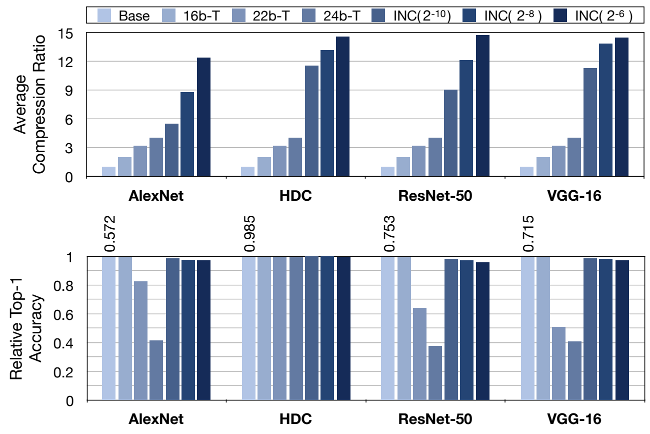


Fig. 14: Comparison of (a) compression ratio and (b) impacts on prediction accuracy of DNNs trained by INCEPTIONN training algorithm with various lossy compression schemes. Note that the accuracy is based on the same epochs of training (without extra epochs) for each model. ("Base" denotes the baseline without compression. The number on top of each "Base" bar denotes the absolute prediction accuracy. xb-T represents truncation of x LSBs. "INC" bars are the results of INCEPTIONN lossy compression with a given error bound.)

TABLE III: The bitwidth distribution of compressed gradients. The compressed gradients are composed of two bits for indication tag and compressed data bits (0, 8, 16, or 32 bits).

		2-bit	10-bit	18-bit	34-bit
AlexNet	INC(2^{-10})	74.9%	3.9%	21.1%	0.1%
	INC(2^{-8})	82.5%	14.8%	2.6%	0.1%
	INC(2^{-6})	93.0%	7.0%	0.0%	0.1%
HDC	INC(2^{-10})	92.0%	6.5%	1.5%	0.0%
	INC(2^{-8})	95.7%	3.4%	0.9%	0.0%
	INC(2^{-6})	98.1%	1.6%	0.4%	0.0%
ResNet-50	INC(2^{-10})	81.6%	17.9%	0.5%	0.0%
	INC(2^{-8})	92.3%	7.7%	0.1%	0.0%
	INC(2^{-6})	97.6%	2.4%	0.0%	0.0%
VGG-16	INC(2^{-10})	94.2%	0.9%	4.9%	0.0%
	INC(2^{-8})	96.2%	3.8%	0.0%	0.0%
	INC(2^{-6})	97.3%	2.7%	0.0%	0.0%

of gradients and INCEPTIONN compression with the absolute error bound of 2^{-10}, 2^{-8} and 2^{-6}. We observe that the naïve truncation of floating-point values only provides low constant compression ratios (*i.e.*, $4\times$ at most) while suffering from substantial accuracy loss (*i.e.*, up to 62.4% degradation in prediction accuracy of trained ResNet-50). This is due to the fact that the compression errors introduced by naïve truncation are uncontrolled and open ended. Moreover, the potential of truncation is limited by the length of the mantissa. Dropping more bits will perturb the exponent (*e.g.*, "24b-T" in Fig. 14), which results in a significant loss of accuracy of trained DNNs. In general, the truncation methods are only suitable for simpler DNNs such as HDC and are not suitable for complex DNNs such as AlexNet, VGG-16, or ResNet-50.

In contrast, our lossy compression shows much higher compression ratios (*i.e.*, up to $14.9\times$) as well as better preserves the training quality than the truncation cases even for those complex DNNs. Fig. 14 shows that the errors induced by compression are well controlled by our algorithm and the average compression ratios are boosted by the relaxation of a given error bound. With the most relaxed error-bound (2^{-6}), almost all benchmarks demonstrate a compression ratio close to $15\times$ and the final accuracies of trained DNNs are only degraded slightly, *i.e.*, $< 2\%$ in absolute accuracy. Note that this slight drop of accuracies incurs only when DNNs are trained for the same number of epochs as their lossless baselines and such drop can be easily compensated by negligible extra epochs of training, as discussed in Sec. VIII-B.

To further understand the significant gains from our compression algorithm, we analyze the bitwidth distribution of compressed gradients. Table III reports the collected statistics. When the error bound is 2^{-6}, for all the evaluated models, our algorithm compresses larger than 90% of gradients into two-bit vectors. Even with 2^{-10} as the error bound, 75% to 94% of gradients are compressed into the two-bit vectors. Leveraging this unique value property of gradients, our lossy compression algorithm achieves significantly larger compression ratio than general-purpose compression algorithms.

Lastly, we find that the compression ratio of the gradients is not necessarily proportional to the reduction in communication time, as shown in Fig. 12 where the compression with an error bound of 2^{-10} should have compressed the communication

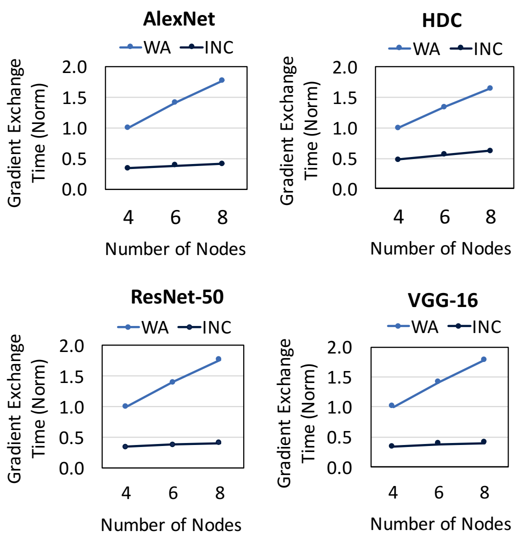


Fig. 15: Scalability of INCEPTIONN training algorithm (INC) as compared to the conventional worker-aggregator based algorithm (WA) with different number of worker nodes. Gradient exchange time consists of both gradient/weight communication and gradient summation time. All values are normalized against four-node WA case.

time by a factor of $5.5 \sim 11.6\times$. This is because we do not reduce the total number of packets and the network stack overhead such as sending network packet headers remains the same. Consequently, the use of more relaxed error bounds (e.g., 2^{-8} and 2^{-6}) only provides marginally additional reduction in the overall communication time.

D. Scalability Evaluation of INCEPTIONN Training Algorithm

We also evaluate the scalability of our INCEPTIONN training algorithm by extending our cluster up to eight worker nodes. Since we had only four GPUs available at our disposal, we only measured the gradient exchange time for the scalability experiments. The gradient exchange time consists of both gradient/weight communication and gradient summation time, and represents the metric in the scalability evaluation, because only communication and summation overheads scale with the number of nodes, while the time consumed by other DNN training steps such as forward pass, backward pass, weight update are constant due to their local computation nature.

Fig. 15 compares the gradient exchange time between the INCEPTIONN baseline (**INC**) and the worker-aggregator baseline approach (**WA**), both without compression across different number of worker nodes. As shown in Fig. 15, the gradient exchange time increases almost linearly with the number of worker nodes in the WA cluster; however, it remains almost constant in the INCEPTIONN cluster, especially when training larger models such as AlexNet, VGG-16, and ResNet-50 where the network bandwidth is the bottleneck. This phenomenon seems intuitive, since in WA cluster both the communication and summation loads congest the aggregator node, while the INCEPTIONN approach balances these two loads by distributing them evenly among worker nodes. Analytically, by adopting the communication models in [24], the gradient exchange time in a WA cluster is: $(1+log(p))\cdot\alpha+(p+log(p))\cdot n\cdot\beta+(p-1)\cdot n\cdot\gamma$, where p denotes the number of workers, α the network link latency, n the model size in bytes, β the byte transfer time, and γ the byte sum reduction time. In practice, for distributed DNN training, the first term is negligible compared to the second and third term, due to the large model size n and the limited network bandwidth β. The above equation explains clearly why the conventional WA approach is not scalable with increasing number of nodes p, *i.e.*, the gradient exchange time is linear in cluster size. In contrast, the communication-balanced INCEPTIONN offers the gradient exchange time of: $2(p-1)\cdot\alpha+2(\frac{p-1}{p})\cdot n\cdot\beta+(\frac{p-1}{p})\cdot n\cdot\gamma$, where the effect of large cluster size p cancels in the second and third term, making INCEPTIONN much more scalable.

IX. Related Work

Acceleration for ML. There has been a large body of work that leverage specialized accelerators for machine learning. Most of the work have concentrated on the inference phase [22, 54–70] while INCEPTIONN specifically aims for accelerating the training phase. Google proposes the TPU [22], which is an accelerator with the systolic array architecture for the inference of neural networks. Microsoft also unveiled Brainwave [68] that uses multiple FPGAs for DNN inference. Eyeriss is also an accelerator for CNN inference of which compute units set a spatial array connected through the reconfigurable multicast on-chip network to support varying shape of CNNs and maximize data reuse.

While the inference phase has been the main target of ML acceleration, the community has recently started looking into the acceleration of training phase [13, 71–75]. ScaleDeep [72] and Tabla [73] are ASIC and FPGA accelerators for the training phase while offering higher performance and efficiency compared to GPUs, which are the most widely used general-purpose processors for ML training. Google Cloud TPU [71] is the next-generation TPU capable of accelerating the training computation on the Google's distributed machine learning framework, Tensorflow [76]. CoSMIC [13] provides a distributed and accelerated ML training system using multiple FPGA or ASIC accelerators. Others [74, 75] focus on the acceleration of neural nets training with approximate arithmetic on FPGA. These ML training accelerators are either single-node solutions or accelerators deployed on the centralized training systems based on worker-aggregator approach, while INCEPTIONN provides a decentralized gradient-based training system and an efficient in-network gradient compression accelerators.

Distributed training algorithms. Li *et al.* [78, 79] proposed a worker-aggregator based framework for distributed training of deep nets and a few approaches to reduce the cost of communication among compute nodes. More specifically, they first explored the key-value store approach that exchanges nonzero weight values, leveraging the sparsity of the weight matrix. Secondly, they adopted a caching approach to reduce the number of key lists that need to be transmitted by caching repeatedly used key lists on both the sending and receiving compute nodes. Third, they deployed approaches that randomly or selectively skip some keys and/or values. Note that these approaches assume that (1) the weights are indeed sparse and (2) the framework updates and maintains weights using centralized nodes. In contrast, our work is a gradient-centric framework that exchanges only gradient values among compute nodes to update each weight, exploiting our observation that the gradient values are much more tolerant to more aggressive lossy compression than weight values. Consequently, our framework efficiently supports dense weights without notably compromising the accuracy of the training procedure.

Recently, Iandola *et al.* [6] diverted from the worker-aggregator architecture and developed a reduction tree based approach. More specifically, instead of workers directly communicating with the aggregators, gradients are reduced by employing a tree based topology. Despite this topology, a central unit remains, which takes care of the weight vectors. Importantly, Iandola *et al.* [6] did not change the communication paradigm, *i.e.*, transmitting gradients to a central unit which then broadcasts the weights.

Similar to the aforementioned algorithms, HogWild! [80], DistBelief [1] and SSP [81] also take the worker-aggregator approach, while they perform asynchronous update of gradients during training to reduce synchronization overhead. These works not only need to deal with stale gradient update, but also significantly rely on centralized aggregators.

Gradient reduction techniques. There has been a series of work that proposes techniques for gradient reductions [12, 25, 26, 82, 83]. Quantization techniques for gradients [25, 26, 82, 83] provide algorithmic solutions to reduce the gradient precision while preserving the training capability. Deep Gradient Compression [12] is a complementary approach that reduces the amount of communication by skipping the communication of the gradients in each iteration. It will only communicate the gradients if the locally accumulated gradient exceeds a certain threshold. These works do not change the worker-aggregator nature of distributed training, nor propose in-network acceleration of compression.

X. Conclusion

Communication is a significant bottleneck in distributed training. The community has pushed forward to address this challenge by offering algorithmic innovations and employing the higher speed networking fabric. However, there has been a lack of solution that conjointly considers these aspects and provides an interconnection infrastructure tailored for distributed training. INCEPTIONN is an initial step in this direction that co-design hardware and algorithms to (1) provide an in-network accelerator for the lossy compression of gradients, and (2) maximize its benefits by introducing the gradient-centric distributed training. Our experiments demonstrate that INCEPTIONN reduces the communication time by 70.9$\sim$80.7% and offers 2.2$\sim$3.1$\times$ speedup over the conventional training system, while achieving the same level of accuracy.

Acknowledgment

This work is supported in part by grants from NSF (CNS-1705047 and CNS-1557244) and SRC/JUMP Applications Driving Architectures (ADA) Research Center, and equipment donations from IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) and Intel.

References

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[2] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *NIPS*, 2013.

[3] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *OSDI*, 2014.

[4] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[5] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training deep networks in spark," in *ICLR*, 2016.

[6] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, "Firecaffe: near-linear acceleration of deep neural network training on compute clusters," in *CVPR*, 2016.

[7] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training ImageNet in 1 hour," in *arXiv:1706.02677 [cs.CV]*, 2017.

[8] S. L. Smith, P. Kindermans, and Q. V. Le, "Don't decay the learning rate, increase the batch size," in *ICLR*, 2018.

[9] Y. You, Z. Zhang, C. Hsieh, and J. Demmel, "100-epoch ImageNet Training with AlexNet in 24 Minutes," in *arXiv:1709.05011v10 [cs.CV]*, 2018.

[10] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, 2015.

[11] F. Iandola, "Exploring the Design Space of Deep Convolutional Neural Networks at Large Scale," in *arXiv:1612.06519*, 2016.

[12] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *ICLR*, 2018.

[13] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *MICRO*, 2017.

[14] D. S. Banerjee, K. Hamidouche, and D. K. Panda, "Re-designing CNTK Deep Learning Framework on Modern GPU Enabled Clusters," in *CloudCom*, 2016.

[15] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters," in *PPoPP*, 2017.

[16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[17] A. Krizhevsky, I. Sutskever, , and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," in *IJCV*, 2015.

[20] I. Kokkinos, "Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory," in *CVPR*, 2017.

[21] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[23] S. Sardashti, A. Arelakis, and P. Stenstrm, *A Primer on Compression in the Memory Hierarchy*. Morgan & Claypool Publishers, 2015.

[24] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich." *IJHPCA*, vol. 19, 2005.

[25] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns." in *INTERSPEECH*, 2014.

[26] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," *CoRR*, vol. abs/1705.07878, 2017.

[27] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *NIPS*, 2017.

[28] N. Dryden, T. Moon, S. A. Jacobs, and B. V. Essen, "Communication quantization for data-parallel training of deep neural networks." in *MLHPC@SC*. IEEE Computer Society, 2016.

[29] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing." in *INTERSPEECH*, 2015.

[30] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 2016.

[31] Google, "Snappy compression: https://github.com/google/snappy," 2011.

[32] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016.

[33] Xilinx INC, "Xilinx virtex-7 fpga vc709 connectivity kit, https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html," 2014.

[34] Xilinx INC, "Virtex-7 fpga xt connectivity targeted reference design for the vc709 board, https://www.xilinx.com/support/documentation/boards_and_kits/vc709/2014_3/ug962-v7-vc709-xt-connectivity-trd-ug.pdf," 2014.

[35] Network Working Group, "Requirement for comments: 3168, https://tools.ietf.org/html/rfc3168," 2001.

[36] M. Alian, A. H. Abulila, L. Jindal, D. Kim, and N. S. Kim, "Ncap: Network-driven, packet context-aware power management for client-server architecture," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017.

[37] Apache Incubator, "Handwritten digit recognition, https://mxnet.incubator.apache.org/tutorials/python/mnist.html," 2017.

[38] Aymeric Damien, "Tensorflow-examples, https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/multilayer_perceptron.py," 2017.

[39] Google INC, "Keras examples, https://github.com/keras-team/keras/blob/master/examples/mnist_mlp.py," 2017.

[40] Krzysztof Sopya, "Tensorflow mnist convolutional network tutorial, https://github.com/ksopyla/tensorflow-mnist-convnets," 2017.

[41] Google INC, "Tensorflow model zoo, https://github.com/tensorflow/models," 2017.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *IEEE*, 1998.
[43] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2010.
[44] INTEL Corporation, "Intel math kernel library, https://software.intel.com/en-us/mkl," 2018.
[45] OpenMPI Community, "Openmpi: A high performance message passing library, https://www.open-mpi.org/," 2017.
[46] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2015.
[47] NVIDIA Corporation, "Nvidia titan xp, https://www.nvidia.com/en-us/design-visualization/products/titan-xp/," 2017.
[48] INTEL Corporation, "Xeon cpu e5, https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors.html," 2017.
[49] Samsung Corporation, "Samsung ddr4, http://www.samsung.com/semiconductor/global/file/product/DDR4-Product-guide-May15.pdf," 2017.
[50] NETGEAR Corporation, "Prosafe xs712t switch, https://www.netgear.com/support/product/xs712t.aspx," 2017.
[51] Alexey Andreyev, "Introducing data center fabric, the next-generation facebook data center network, https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/," 2014.
[52] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hlzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in googles datacenter network," in *Sigcomm '15*, 2015.
[53] INTEL Corporation, "Intel x540, https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x540-t2-brief.html," 2017.
[54] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[55] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *ISCA*, 2017.
[56] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.
[57] P. Judd, A. Delmas, S. Sharify, and A. Moshovos, "Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing," in *ISCA*, 2016.
[58] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.
[59] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.
[60] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.
[61] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: shifting vision processing closer to the sensor," in *ISCA*, 2015.
[62] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer." in *MICRO*, 2014.
[63] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Misra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *MICRO*, Oct. 2016.
[64] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *HPCA*, 2017.
[65] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerator," in *MICRO*, 2016.
[66] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO*, 2017.
[67] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.
[68] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating persistent neural networks at datacenter scale," in *HotChips*, 2017.
[69] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen, "Machine learning on FPGAs to face the IoT revolution," in *Proceedings of the 36th International Conference on Computer-Aided Design*, ser. ICCAD '17. IEEE Press, 2017.
[70] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, 2017.
[71] J. Dean and U. Hölzle, "Google Cloud TPUs," https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/, 2017.
[72] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA*, 2017.
[73] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *HPCA*, 2016.
[74] Q. Wang, Y. Li, and P. Li, "Liquid state machine based pattern recognition on fpga with firing-activity dependent power gating and approximate computing," in *ISCAS*, 2016.
[75] Q. Wang, Y. Li, B. Shao, S. Dey, and P. Li, "Energy efficient parallel neuromorphic architectures with approximate arithmetic on fpga." *Neurocomputing*, vol. 221, 2017.
[76] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467 [cs]*, 2016.
[77] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.
[78] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[79] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, “Communication efficient distributed machine learning with the parameter server,” in *NIPS*, 2014.

[80] B. Recht, C. R, S. J. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent.” in *NIPS*, 2011.

[81] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, “More effective distributed ml via a stale synchronous parallel parameter server.” in *NIPS*, 2013.

[82] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, “QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding,” *arXiv:1610.02132 [cs]*, 2017.

[83] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients,” *arXiv:1606.06160 [cs]*, 2016.