SqueezeNIC: Low-Latency In-NIC Compression for Distributed Deep Learning

Achref Rebai
KAUST

Mubarak Adetunji Ojewale
KAUST

Anees Ullah
University of Engineering and Technology Peshawar

Marco Canini
KAUST

Suhaib A. Fahmy
KAUST

Abstract

To alleviate the communication bottleneck of distributed deep learning training, several data compression algorithms have been proposed. However, these algorithms introduce computational overhead and resource allocation concerns on CPUs and GPUs. In this paper, we introduce SqueezeNIC, an FPGA-based Network Interface Card (NIC) that offloads communication compression from CPUs/GPUs, bridging a high bandwidth intra-node network with a high bandwidth inter-node network. It enables better overlap of gradient communication and computation to further reduce training time per iteration in distributed training. Our evaluations shows that SqueezeNIC achieves line rate compression and can speed up training by up to a factor of 1.21×, compared to baseline approaches.

CCS Concepts

• **Networks → In-network processing**.

Keywords

In-Network Compression, Distributed Training, FPGA

ACM Reference Format:
Achref Rebai, Mubarak Adetunji Ojewale, Anees Ullah, Marco Canini, and Suhaib A. Fahmy. 2024. SqueezeNIC: Low-Latency In-NIC Compression for Distributed Deep Learning. In *SIGCOMM Workshop on Networks for AI Computing (NAIC '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3672198.3673801

1 Introduction

Distributed Deep Learning (DDL) has experienced rapid growth in recent years, due in part to the emergence of very large deep learning models on the one hand and the ever-increasing size of training data on the other hand [1, 13, 14, 16, 21, 25, 27, 28]. The Data Parallel (DP) distributed training paradigm is typically adopted when the size of a Deep Neural Network model fits within the memory of a single compute worker [56], where the model is replicated across multiple workers, with training data partitioned among them. Weight and gradient aggregation among workers occurs at periodic intervals, facilitated by collective communication primitives. Conversely, when the model exceeds the memory capacity of a single worker, various distributed training methodologies can be employed, including Pipeline Parallel [26], Tensor Parallel [43], and Fully Sharded Data Parallel (FSDP) [38, 58] approaches. In all of these methodologies, the model is partitioned among multiple workers, with intermediate results exchanged along with weight and gradient aggregation similar to DP.

All of these distributed training strategies are based on collective communication operations to synchronize intermediate results between several workers during training. Although DDL has greatly improved training throughput, weight and gradient communication have become a performance bottleneck [40, 42, 54]. Within a node, GPUs can synchronize between training steps over high-speed internal networks such as NVLink, which provides 600-900 GB/s bi-directional bandwidth [34]. This enables efficient communication and coordination between GPUs, enabling fast collective operations within a node. However, for collective operations involving multiple nodes, communication over the inter-node network becomes a bottleneck because inter-node network bandwidth of 50-100 GB/s [60, 61], is an order of magnitude lower than intra-node.

To address this mismatch, Data Compression (DC) [32] algorithms have been proposed for inter-node communication in DDL. These can be divided into three main classes, namely: (1) Sparsification algorithms - where sparsity in the gradient/weight matrix is exploited and only a subset of values are transmitted [5, 32, 52]; (2) Quantization algorithms that limit the numerical precision used to represent gradients [7, 11, 30, 53, 54], and finally (3) Low-Rank decomposition [45, 47, 48]. General-purpose CPUs are not well suited to DC due to their limited computational power and the bottleneck imposed by the PCIe interface. On the other hand, while GPUs are well suited for DC operations, they incur a non-negligible computational overhead and consume computational resources that are normally reserved for training and other computational tasks. Therefore, using GPUs for data compression can slow down the training process [4, 50].

In this work, we propose to offload the compression operations to an FPGA-based NIC, named *SqueezeNIC*. SqueezeNIC enables compression operations with negligible overhead on the one hand, as well as further reducing compression overhead by overlapping compression operations with communication. Taken together these factors mean SqueezeNIC reduces training time per iteration in the DDL environment. Crucially, SqueezeNIC is designed to interface a high bandwidth intra-node GPU network with a traditional

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
NAIC '24, August 4–8, 2024, Sydney, NSW, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0713-1/24/08
https://doi.org/10.1145/3672198.3673801

datacenter inter-node network, to overcome the significant bandwidth mismatch between them. Specifically, we make the following contributions: 1) We present the concept, design, and implementation of an FPGA-based NIC, called SqueezeNIC, that provides high computational throughput for DC, and novel approaches for application in DDL training. 2) We experimentally demonstrate that SqueezeNIC can perform compression operations at line-rate with negligible overhead, resulting in a speedup of 1.21× compared to baseline approaches.

2 Design specification

SqueezeNIC is a hardware component that connects an intra-node aggregation network to an inter-node network. The intra-node network is a high-bandwidth on-board interconnect enabling multiple GPUs to communicate at high bandwidth, such as NVLink. The inter-node network is a high-speed Ethernet network built using commodity hardware components. The key aim of SqueezeNIC is to use compression to overcome the bandwidth imbalance between the intra-node and inter-node networks. For example, NVLink 3.0 offers an aggregate bi-directional bandwidth of 600 GB/s, while NVLink 4.0 offers 900 GB/s. These bandwidths are over an order of magnitude higher than the 50 GB/s (400 Gbps) networks currently available. SqueezeNIC uses compression to mimic a higher throughput inter-node network to allow for faster collective communication in distributed applications.

To enable compression with negligible overhead, SqueezeNIC is required to ingest data at the extremely high intra-node bandwidth, complete compression operations at line-rate and transmit compressed data at the high inter-node bandwidth. This requires hardware acceleration through deep pipelines that can parallelize and pipeline compression operations and meet the required data rates. For connectivity to both the intra-node network and the inter-node network, high frequency transceivers are required. These are used on all platforms that support high speed serial interface standards, such as PCI Express, High Bandwidth Memory, or NVMe. Considering the bandwidths under consideration, a sufficiently high number of transceiver pairs are required to support the required rates at the intra- and inter-node sides.

2.1 Architecture

In selecting a suitable hardware architecture, we seek to balance the high bandwidth and computational requirements with the fact that proven approaches to compression in distributed deep learning are diverse. Indeed, different models benefit from different types of compression [57], and even as training progresses, different forms of compression can be beneficial [2, 3, 6, 39, 55]. Hence, we also require computational flexibility. Finally, a key factor in the types of computation required for compression is efficient bit-level manipulation, especially in the case of quantization. Hence, we propose using FPGAs as our deployment platform. We show that they are able to (1) sustain the required interconnect bandwidth, (2) meet the resulting processing requirements through sufficient parallelization and pipelining, and (3) entail minimal latency.

In terms of connectivity, FPGAs offer ample numbers of high speed transceiver pairs. On commercially available FPGA accelerator boards, these are used to interface FPGAs with PCI Express and high-speed networking. Mid-size and larger devices in the Xilinx Virtex UltraScale+ family contain 80-100+ transceivers, each supporting 32.75 Gbps communication, for an aggregate bandwidth of 2.6-4+ Tbps into and out of the FPGA. This makes FPGAs suitable to interface directly with the intra-node network (e.g., by communicating directly with GPUs via NVLink).

Quantization acceleration. Quantization is a point-wise operation which can be applied to a stream without intermediate packet-level buffering. To compress from host-to-network, SqueezeNIC ingests the host side stream from the intra-node network at line rate, and applies the point-wise compression operation to the tensor values in this stream with sufficient parallelization to maintain line rate processing. Depending on the quantization operation being performed, the depth of the processing pipeline can vary from a few to tens of clock cycles. This added latency is negligible in terms of impact on overall communication time. Packets for transmission are assembled inside SqueezeNIC and sent out on the inter-node network. To decompress from network-to-host, network packets are ingested from the network at line rate and decompressed through parallel decompressors and reassembled into the packet format for the intra-node network. Any metadata required for the quantization operation is assumed to be supplied by the GPUs in a packet header.

Sparsification Acceleration Sparsification is more complex. If the sparsification operation can be expressed through a streaming process using metadata from the GPUs (such as sending only tensor values above a fixed threshold value, or block-based sparsification [19]), then the compression operators are laid out as for the quantization mode above. However, if the sparsification is a multi-pass operation, then SqueezeNIC is required to buffer tensors (or blocks) internally. To enable line-rate processing, HBM memory is used with a double buffered arrangement. The first side of the double buffer is filled by the host intra-node link, while the second side is read into the sparsifier architecture and out over the inter-node network. In this scenario, the performance of SqueezeNIC is bottlenecked by the bandwidth to HBM. Currently, the HBM2 on the AMD Virtex UltraScale+ offers an aggregate bandwidth of 460 GB/s, while the HBM2e on the AMD Versal ACAP offers an aggregate bandwidth of 819 GB/s. With the double-buffer, the bottleneck would be half of this bandwidth. HBM3 would be sufficient to support the currently available NVLink bandwidths for arbitrary sparsification operations.

2.2 SqueezeNIC Compression Approaches

For simplicity, we employ the Data Parallel (DP) distributed training approach in describing SqueezeNIC compression approaches. However, we note that compression is also applied in other strategies such as FSDP [46]. In DP, only the weight gradients are communicated; therefore, compression is discussed in the context of gradient compression. Prior works [10, 57] usually first perform a local aggregation of the gradient vectors across the intra-node GPUs, then a GPU within the node compresses the aggregated gradient vector. Afterwards, inter-node aggregation is invoked, where the inputs consist of a single compressed gradient vector per node. Finally, each GPU obtains a copy of the aggregated gradient and decompresses it. We view this approach as the simplest *baseline* for our

work. Although there are many ways to perform compression operations with respect to intra-node communication, we consider only two approaches where we believe we can intuitively perform better than the baseline. We discuss the two approaches below.

Approach 1: Local Reduction + In-Network Compression (LR+INC). In this approach, a local reduce-scatter operation takes place among the intra-node GPUs. After this operation, each of the GPUs transmits the different partitions of the reduced gradients that it holds to SqueezeNIC. Here, SqueezeNIC compresses the received local gradient, initiates a global reduction with other nodes, and subsequently broadcasts the resultant aggregated gradient to all local GPUs.

Approach 2: Compress, Then Local Reduction + In-Network Compression (CTLR+INC). Another approach follows a similar pattern to the previous approach, but with a slight variation. In this scenario, each GPU first compresses its own gradient before any local aggregation occurs. Once each GPU has compressed its gradient, these compressed gradients are then locally aggregated within each node. To avoid format conversion overheads, this step requires that the compressed gradients can be aggregated in their compressed format (e.g., as done in [54]). Afterward, the aggregated gradient from each node is sent to SqueezeNIC. SqueezeNIC performs further compression on the already aggregated gradients and facilitates global reduction across different nodes. The final aggregated gradient is then broadcast to all GPUs in the network, completing the optimization process.

Further optimizations. To reduce latency, we use pipelining to optimize the above approaches. To apply pipelining, the input gradient vector is divided into K partitions; we split the pipeline into two stages: one for gradient computation and the other for gradient synchronization. The synchronization stage includes gradient communication, compression, and decompression operations. Pipelining allows us to overlap both stages while one partition is being computed another can undergo synchronization improving the throughput of the training process and node resource utilization.

3 Performance model

We now analyze the proposed approaches in terms of the number of computational and communication operations and the duration of each of these operations per iteration. We assume N nodes interconnected by a non-blocking inter-node network with full-bisection bandwidth. Each node is connected to the fabric with an effective bandwidth B in each direction (transmit and receive). Each node has M GPUs interconnected via internal links with an effective bandwidth V in each direction. The original gradient vector has a size of z bytes on each GPU. Gradient compression uses a compression rate r so that the compressed gradient has size rz.

We assume that all approaches have the same backpropagation time, which allows us to ignore this in the performance comparison. Additionally, to compare the proposed approaches with each other (not with the baseline), we can also ignore the inter-node reduction time since the process is the same for all approaches. We assume that the communication routine uses ring AllReduce for gradient reduction and we follow the modeling approach of Patarasuk et al. [35]. The AllReduce algorithm consists of two phases; the ReduceScatter and the AllGather phases. For intra-node reduction, our proposed approaches only require the ReduceScatter phase and does not have to broadcast the aggregated gradient to all the GPUs inside the node. Instead, they only need to send the aggregated gradient for global (inter-node) reduction.

Table 1: Definition of notation. Both GPUs and SqueezeNIC are termed accelerators.

Term	Description
$g_{i,k}$	Gradient partition k at training iteration i
$T^{intra}_{comm}(g_{i,k})$	Time taken to communicate gradient $g_{i,k}$ between two accelerators
$T^{inter}_{comm}(g_{i,k})$	Time taken to communicate gradient $g_{i,k}$ between two accelerators
$T^{Hw}_{Comp}(g_{i,k})$	Time taken to compress gradient $g_{i,k}$ on a hardware accelerator
$T^{Hw}_{Decom}(Q(g_{i,k}))$	Time taken to decompress gradient $Q(g_{i,k})$ on a hardware accelerator
$Q(g_{i,k})$	Compressed gradient using compression operator $Q(.)$

We employ the alpha-beta communication model [35] to estimate the communication costs. The model considers "alpha" as the latency overhead–the fixed time cost of initiating a communication–and "beta" as the inverse bandwidth, representing the time required to transmit each unit of data.

$$T(z) = \alpha + \beta z$$

We also introduce the necessary terms to simplify the analysis and effectively capture the performance components in Table 1.

Having introduced all the necessary notation, we proceed to analyze the performance of the proposed approaches. We start by modeling the time needed for the baseline. The gradient synchronization baseline starts with a compressed local reduction using GPUs followed by a compressed global reduction using GPUs and NICs for communication.

$$T_{baseline}(g_{i,k}) = (M-1)T^{intra}_{comm}(g_{i,k}) + 2(N-1)T^{inter}_{comm}(g_{i,k}) \quad (1)$$

Approach 1 (ReduceScatter for the local reduction, then Ring AllReduce): The first approach starts with non-compressed local reduction, then the locally aggregated gradient is sent to the hardware accelerator (SqueezeNIC or GPU) to perform a compressed global reduction.

$$\begin{aligned} T_{App1}(g_{i,k}) =& (M-1)T^{intra}_{comm}(g_{i,k}) + 2(N-1)T^{inter}_{comm}(Q(g_{i,k})) \\ &+ N \cdot T^{Hw}_{Comp}(g_{i,k}) + N \cdot T^{Hw}_{Decom}(Q(g_{i,k})) \end{aligned} \quad (2)$$

Approach 2 (ReduceScatter with compression for the local reduction then Ring AllReduce): The second approach is similar to the first but this time the local reduction is performed with compression.

$$\begin{aligned} T_{App2}(g_{i,k}) =& (M-1)T^{intra}_{comm}(Q(g_{i,k})) + (M-1)T^{GPU}_{Comp}(g_{i,k}) \\ &+ (M-1)T^{GPU}_{Decom}(Q(g_{i,k})) + N \cdot T^{Hw}_{Decom}(Q(g_{i,k}) \\ &+ N \cdot T^{Hw}_{Comp}(g_{i,k}) + 2(N-1)T^{inter}_{comm}(Q(g_{i,k}))) \end{aligned} \quad (3)$$

With this model, we simulate and analyze the total communication time for (1) a broadcast operation from 1 GPU to 8 other GPUs and an (2) inter-node AllReduce operation between 8 different nodes. For (1), we performed our experiment on a node with 8 V100 GPUs. The node uses a hybrid 8-GPU hybrid cube-mesh network as in the DGX1 (with Tesla V100) system topology. This topology ensures high bandwidth intra-node communication. For collective AllReduce inter-node communication (2), we use a set of nodes with a V100 GPU connected to an Mellanox ConnectX 5 NIC. In both experiments, we use PyTorch 2.0.1 and NCCL 2.14.3. We monitor the inter-node AllReduce time for different node counts from 2 to 16 to have a model that allows us to scale the number of workers up to 16.

Fig. 1 and Fig. 2 show the results for scenarios (1) and (2), respectively. It can be seen from the figures that the values obtained with the model match the measured values of the actual system, with an error margin of less than 10%. Having established the accuracy of the performance model, we use this model to quantitatively evaluate the performance of SqueezeNIC in distributed end-to-end training in Section 4.4.

4 Experiments

We demonstrate the feasibility and effectiveness of our proposal. First, in Section 4.1 and Section 4.2, we show that performing compression on GPUs or DPUs incurs significant overhead (justifying the need for FPGA offloading). In Section 4.3, we show that our SqueezeNIC FPGA implementation can perform line-rate compression, even at a network speed of 400 Gbps. Finally, in Section 4.4, we show the advantage of SqueezeNIC in terms of training throughput improvements. Again, for simplicity, we employ the DP distributed training approach in all our experiments.

4.1 In-GPU and In-DPU Gradient Compression

Setup. We conduct our experiments on two different machines: one for GPU evaluation and the other for DPU (a programmable SmartNIC) evaluation. The GPU machine is equipped with NVIDIA Tesla V100 GPUs (32 GB memory) and an Intel Xeon Gold 6242 CPU @ 2.80 GHz, running CentOS Linux 7 with CUDA 10.1, cuDNN 7.5.1, MXNet 1.5.0, and PyTorch 1.10.2. The DPU machine uses a Mellanox BlueField-2 equipped with 8 Arm v8 A72 cores (64-bit), running Ubuntu 20.04.3 LTS arch64 with PyTorch 1.11.0.

Baseline. We first extract gradient tensors from four different DNN models: BERT [18], DeepLight [17], NCF [24], and ResNet [23]. Using the GPU implementation of compression algorithms from Hipress [10], we measure the average time to compute gradient tensor compression for each layer on the GPU. The five GPU compression algorithms evaluated are Terngrad [53], DGC [32], TBQ [44], GradDrop [5], and 1-Bit [41]. We then profile each compression algorithm's GPU kernel utilization. For DPU compression, we use the Grace implementation [57] for Terngrad and DGC compression algorithms, running them on BlueField-2 DPU while monitoring CPU utilization.

Results. Table 2 shows that, as expected, compression on GPU outperforms compression on DPU for all cases. The speedup of GPU tensor compression over the DPU ranges from 3× for ResNet with DGC to 187× for NCF with Terngrad. For the compute throughput,

Table 2: Time to compress one model gradient tensor in seconds.

	Terngrad		DGC		TBQ	GradDrop	1-Bit
Model	GPU	DPU	GPU	DPU	GPU	GPU	GPU
BERT	0.122	6.065	0.180	3.196	0.154	0.190	0.062
NCF	0.004	0.756	0.006	0.248	0.005	0.006	0.002
DeepLight	0.026	0.298	0.034	0.148	0.029	0.036	0.013
ResNet	0.097	0.866	0.133	0.509	0.106	0.140	0.035

Table 3: GPU compute throughput for each gradient compression algorithm.

Algorithm	Terngrad	DGC	TBQ	GradDrop	1-Bit
Av. Comp. T'put(%)	29.85	6.608	21.84	6.931	76.61

Table 4: Iteration time per batch

	Compression Scheme					
Batch training Time (s)	None	Terngrad	DGC	TBQ	GradDrop	1-Bit
DeepLight	0.024	0.113	0.126	0.130	0.129	0.106
BERT	3.300	7.734	N/A	7.926	N/A	7.396

Table 3 shows that the average GPU utilization for each compression algorithm ranges from 6% to 76.61%. While for DPU compression, the average CPU utilization is about 50% for each core.

Takeaway. Performing compression operations on the GPU is more efficient than on DPUs because the compression algorithms exploit the parallel nature of the algorithm, which makes the current general-purpose DPUs unsuitable for compression offloading. Furthermore, both GPU and DPU compression do not fully utilize the computing power of the hardware, which shows that there is room for improvement.

4.2 Compression Overhead

Setup. We use the same GPU machine as in Section 4.1.

Baseline. To investigate the impact of compression on the training process, we train DeepLight and BERT models on a single GPU while performing compression and decompression of each gradient tensor for every training iteration while monitoring the time required to process a single batch.

Results. Table 4 shows the different times taken to process a batch for each compression algorithm as compared to the no compression baseline. All four algorithms have an iteration time 4-5× slower than the no compression baseline for DeepLight. For BERT, the iteration time is 2× when we apply compression.

Takeaway. While GPUs are more efficient at handling compression than DPUs, the overhead of GPU compression increases training time, making this approach sub-optimal. This inefficiency motivates the need to offload compression operations to more efficient hardware.

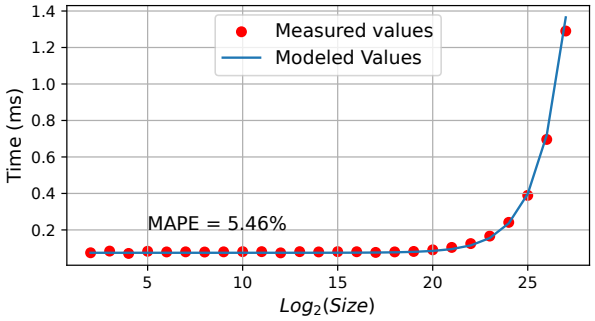

Figure 1: Broadcast model for 8 GPUs.

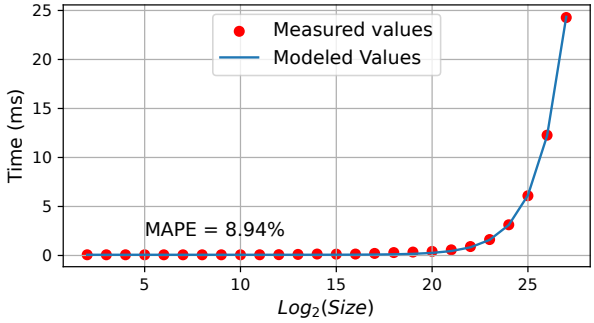

Figure 2: AllReduce model for 8 nodes.

Table 5: Measured host to network FPGA throughput (Gbps) with different compression ratios.

Compression	Packet size		
Ratio	308 B	1076 B	1518 B
0	10.68	42.27	93.19
2	8.00	42.19	92.17
8	7.98	42.66	90.33
32	7.66	42.15	95.22

Table 6: FPGA resource utilization percentage scaling with increasing host-side bandwidth.

Host BW (Gbps)	LUTs	FFs	CLBs
400	11.85	11.28	26.53
800	12.40	11.81	26.91
1600	13.60	12.77	29.03
3200	12.59	12.55	28.07
6400	19.31	19.30	39.97

4.3 SqueezeNIC Line-Rate Compression

Having established that gradient compression, even when done on a GPU or DPU, instills considerable time overhead, we now proceed to demonstrate that SqueezeNIC can perform compression with near zero overhead and at line rate. Our goal with these experiments is to merely provide a proof of concept of our design. At present, we are unable to implement an NVLink connection between the FPGA and the intra-node network; since we are limited by the PCIe bandwidth we also experiment with on-chip data generators.

Setup. We use an AMD/Xilinx Alveo U280 FPGA accelerator card. This offers PCIe Gen 3×8 host connectivity and dual 100 Gbps network interfaces. These interfaces are exposed at the hardware level using AXI-Stream, allowing an accelerator to consume and produce data every clock cycle (250 MHz in our case). We use the Xilinx OpenNIC design to functionally verify our compressors.

Baseline. We develop a library of quantization-based compressors with Vitis HLS running at 250 MHz on the TX path with equivalent decompression on the RX path. For functional verification the FPGA board is connected with a QSFP28 loopback cable and host-side data generation is performed by DPDK pktgen.

Extension. Since our proposal extends the required bandwidth at the host side beyond what PCIe can accommodate, we perform throughput experiments using on-chip data generators to mimic host-side data generation at the higher rates equivalent to intra-node network bandwidth.

Results. Our experiments on the FPGA demonstrate two points. First, results in Table 5 demonstrate that the compressor has negligible effect on network throughput compared to the baseline NIC functionality.

Second, we demonstrate that the FPGA compute is capable of scaling to line rate compression for host-side throughput well beyond 100 Gbps. Multiple 100 Gbps data generators are instantiated on-chip to emulate host-side scaling, for example, in Table 6, 400 Gbps is achieved by 4 parallel generators. All designs are matched with a compression ratio to achieve 100 Gbps network-side throughput, which is the maximum supported by the Alveo U280. It can be seen that the device utilization remains low, even for emulated 6400 Gbps host-side bandwidth, with CLB resource utilization under 40%. For verification purposes, the Alveo-based generators were connected to a BlueField-2 SmartNIC to confirm that 100 Gbps line-rate was achieved. Results for 308 byte and 1076 byte packets were verified with the SwitchML framework [40] and functionally tested in a closed loop to demonstrate the functionality and verify quantization error bounds. Results for 1518 byte packets were generated with DPDK pktgen to maximize host throughput.

Takeaway. The FPGA demonstrates its ability to compress at line rate without affecting bandwidth, while also scaling to the very high host bandwidth proposed for the SqueezeNIC design. While existing boards cannot be deployed in this manner at present, due to the required form-factor, the FPGA devices themselves have sufficient high-bandwidth I/O and on-device computing resources to support such a deployment. Our experiments demonstrate that a suitable hardware platform can be built for this purpose.

4.4 Expected Gains

In Section 3, we studied the training performance model, we depicted the different communication pattern in a training iteration and we modeled the execution time for SqueezeNIC collective communication routines. Afterwards in Section 4.3, we showed the

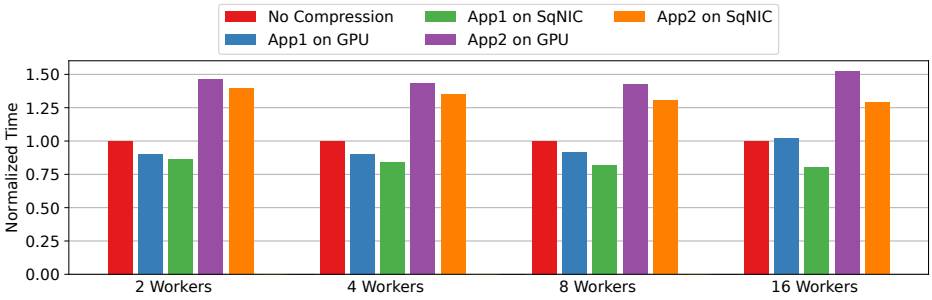

Figure 3: Single batch processing time for BERT-Large model normalized against the no compression baseline.

ability of performing zero cost compression without impacting the bandwidth.

Currently, our SqueezeNIC implementation is confined to a standalone FPGA device and has not yet been integrated with actual servers and accelerators. Consequently, we simulate the expected performance gains using our performance model, integrating the communication model into PyTorch communication hooks. These hooks allows us to customize the gradient communication pattern and integrate every SqueezeNIC communication operation execution time.

We now evaluate the expected performance gains of our proposed approach, focusing on the time taken to process one batch. We test the BERT-Large model with a synthetic dataset using a batch size of 32. We employ weak scaling as we increase the number of workers and set the partition size to 25 MB. The experiment utilizes the 1-Bit quantization method.

Setup. Our testbed consists of a single machine equipped with seven NVIDIA V100 GPUs running PyTorch 2.0.1, CUDA 11.7, and NCCL 2.14.3. To simulate the behavior of multiple nodes, we leverage communication models presented in Section 3 to capture inter-node communication times, enabling us to simulate a multi-node environment.

Baseline. We explore three baselines: 1) *No compression baseline.* 2) *On GPU Approach 1 baseline in direct comparison with Approach 1 on SqueezeNIC.* 3) *On GPU Approach 2 baseline in direct comparison with Approach 2 on SqueezeNIC.*

Results. Figure 3 presents the simulation results for 2, 4, 8, and 16 nodes. The results indicate that SqueezeNIC significantly improves iteration time for both approaches, with notable benefits at 16 nodes. By offloading compression operations from the GPU, SqueezeNIC enhances training speed by 21% for approach 1 and 15% for approach 2 with 16 workers.

Takeaway. As the number of nodes increases, the overhead associated with compression on GPU also rises, often making it difficult to achieve any performance gain compared to a baseline without compression. This challenge is effectively addressed by SqueezeNIC, which implements line-rate compression to enhance performance, particularly when scaling up the number of nodes.

5 Related work

Lin et al. [32] highlight that the computational overhead of compression algorithms in DDL correlates with the compression ratio, with higher ratios leading to higher overhead. Xu et al. [57] and Agrawal et al. [4] find that this overhead can sometimes exceed the communication cost saved, especially in high-bandwidth environments. Bai et al. [10] propose selective compression to mask the DC overhead on GPUs, and Zhong et al. [59] introduce 2-way pipelined DC on CPUs to reduce the overhead, but limited to the parameter server architecture. Wang et al. [51] address intra- and inter-node communication by offloading compression to CPUs, but these methods fail to overlap compression and computation due to resource contention. SqueezeNIC, on the other hand, frees GPUs and CPUs from DC overheads and improves communication and computation overlap.

Alvarez et al. [8] and Tahmasbi et al. [9] propose to offload scatter-gather operations and transport layer logic to FPGA-based SmartNICs, while Spaziani et al. [12] focus on offloading packet processing tasks. Wang et al. [49] propose an FPGA-based SmartNIC for direct P2P communication between GPUs, avoiding the involvement of the CPU. Despite the existence of several commercial SmartNIC solutions [20, 33, 37], none of them is specifically designed for compression operations or the imbalance between intra- and inter-node communication in multi-GPU environments. Other approaches, such as priority-based scheduling [22, 29, 36] and in-network aggregation [31, 40], aim to accelerate DDL training, with De Sensi et al. [15] proposing a switch architecture for user-defined packet handling.

6 Conclusion

We have proposed SqueezeNIC, an FPGA-based compressor NIC to address two major challenges related to DDL training. It addresses the bottleneck caused by the bandwidth imbalance between intra- and inter-node communication. It also relieves the computational nodes (CPUs and GPUs) from the computational overheads of compression that slow down training. Through rigorous experiments, we show that SqueezeNIC can perform line-rate compression even at very high data rates of up to 400 Gbps, with minimal overhead. Importantly, our experiments also show that SqueezeNIC with hierarchical aggregation has the potential to reduce the training time per iteration, thereby increasing throughput compared to baseline approaches. In the future, we plan to investigate how SqueezeNIC features can be extended to further optimize DDL training and inference.

Acknowledgments

This publication is based upon work supported by the King Abdullah University of Science and Technology (KAUST) Office of Research Administration (ORA) under Award No. ORA-CRG2022-5017. For computer time, this research used the resources of the Supercomputing Laboratory at KAUST.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffr ey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.

[2] Ahmed M. Abdelmoniem and Marco Canini. 2021. DC2: Delay-aware Compression Control for Distributed Machine Learning. In *INFOCOM*.

[3] Ahmed M. Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems. In *MLSys*.

[4] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2022. On the utility of gradient compression in distributed training systems. In *MLSys*.

[5] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. In *EMNLP*.

[6] Mohammadreza Alimohammadi, Ilia Markov, Elias Frantar, and Dan Alistarh. 2022. L-GreCo: An Efficient and General Framework for Layerwise-Adaptive Gradient Compression. (2022). arXiv:cs.LG/2210.17357

[7] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Randomized Quantization. In *NeurIPS*.

[8] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. 2020. Specializing the Network for Scatter-Gather Workloads. In *SoCC*.

[9] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *NSDI*.

[10] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel dnn training. In *SOSP*.

[11] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. signSGD: Compressed Optimisation for Non-Convex Problems. In *ICML*.

[12] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *OSDI*.

[13] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. 2020. Elastic Parameter Server Load Distribution in Deep Learning Clusters. In *SoCC*.

[14] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*.

[15] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible in-Network Allreduce. In *SC*.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *NeurIPS*.

[17] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. 2021. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. In *WSDM*.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (2018). arXiv:cs.CL/1810.04805

[19] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning. In *SIGCOMM*.

[20] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*.

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*.

[22] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *MLSys*.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[24] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *WWW*.

[25] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NeurIPS*.

[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[27] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *NSDI*.

[28] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *CVPR*.

[29] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. In *MLSys*.

[30] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *ICML*.

[31] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*.

[32] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*.

[33] Mellanox. 2020. Mellanox innova-2flex. (2020). https://network.nvidia.com/files/doc-2020/pb-innova-2-flex.pdf Online; accessed 14-June-2024.

[34] Nvidia. 2023. NVLink and NVSwitch. (2023). https://www.nvidia.com/en-us/data-center/nvlink/ Online; accessed 14-June-2024.

[35] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009).

[36] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*.

[37] Pensando. 2020. Pensando DSC-25 Distributed Services Card. (2020). https://pensando.my.salesforce-sites.com/DownloadFile?id=a0L4T000004IKurUAG Online; accessed 14-June-2024.

[38] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. (2020). arXiv:cs.LG/1910.02054

[39] Atal Narayan Sahu, Aritra Dutta, Ahmed M. Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. 2021. Rethinking gradient sparsification as total error minimization. In *NeurIPS*.

[40] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *NSDI*.

[41] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs. In *INTERSPEECH*.

[42] Shaohuai Shi, Xianhao Zhou, Shutao Song, Xingyao Wang, Zilin Zhu, Xue Huang, Xinan Jiang, Feihu Zhou, Zhenyu Guo, Liqiang Xie, Rui Lan, Xianbin Ouyang, Yan Zhang, Jieqian Wei, Jing Gong, Weiliang Lin, Ping Gao, Peng Meng, Xiaomin Xu, Chenyang Guo, Bo Yang, Zhibo Chen, Yongjian Wu, and Xiaowen Chu. 2021. Towards Scalable Distributed Training of Deep Learning on Public Cloud Clusters. In *MLSys*.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. (2020). arXiv:cs.CL/1909.08053

[44] Nikko Strom. 2015. Scalable Distributed DNN Training using Commodity GPU Cloud Computing. In *INTERSPEECH*.

[45] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. In *NeurIPS*.

[46] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2024. ZeRO++: Extremely Efficient Collective Communication for Large Model Training. In *ICLR*.

[47] Hongyi Wang, Saurabh Agarwal, and Dimitris Papailiopoulos. 2021. Pufferfish: Communication-efficient models at no extra cost. In *MLSys*.

[48] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018. ATOMO: Communication Efficient Learning

via Atomic Sparsification. In *NeurIPS*.

[49] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *USENIX ATC*.

[50] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2022. ByteComp: Revisiting Gradient Compression in Distributed Training. (2022). arXiv:cs.LG/2205.14465

[51] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *EuroSys*.

[52] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient Sparsification for Communication-Efficient Distributed Optimization. In *NeurIPS*.

[53] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *NeurIPS*.

[54] Jihao Xin, Marco Canini, Peter Richtárik, and Samuel Horváth. 2023. GlobalQSGD: Practical Floatless Quantization for Distributed Learning with Theoretical Guarantees. (2023). arXiv:cs.LG/2305.18627

[55] Jihao Xin, Ivan Ilin, Shunkang Zhang, Marco Canini, and Peter Richtárik. 2023. Kimad: Adaptive Gradient Compression with Bandwidth Awareness. In *DistributedML*.

[56] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data* 1, 2 (2015).

[57] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. 2021. GRACE: A Compressed Communication Framework for Distributed Machine Learning. In *ICDCS*.

[58] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023).

[59] Yuchen Zhong, Cong Xie, Shuai Zheng, and Haibin Lin. 2021. Compressed Communication for Distributed Training: Adaptive Methods and System. (2021). arXiv:cs.DC/2105.07829

[60] Xiang Zhou, Cedric F. Lam, Ryohei Urata, and Hong Liu. 2023. State-of-the-Art 800G/1.6T Datacom Interconnects and Outlook for 3.2T. In *OFC*.

[61] Xiang Zhou, Ryohei Urata, and Hong Liu. 2020. Beyond 1 Tb/s Intra-Data Center Interconnect Technology: IM-DD OR Coherent? *Journal of Lightwave Technology* 38, 2 (2020).