2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems

iRDMA: Efficient Use of RDMA in Distributed Deep Learning Systems

Yufei Ren*, Xingbo Wu†, Li Zhang*, Yandong Wang*, Wei Zhang*, Zijun Wang*, Michel Hack*, Song Jiang†
*IBM T. J. Watson Research Center †University of Texas at Arlington

***Abstract*—Distributed deep learning systems place stringent requirement on communication bandwidth in its model training with large volumes of input data under user-time constraint. The communications take place mainly between cluster of worker nodes for training data and parameter servers for maintaining a global trained model. For fast convergence the worker nodes and parameter servers have to frequently exchange billions of parameters to quickly broadcast updates and minimize staleness. Demand on the bandwidth becomes even higher with the introduction of dedicated GPUs in the computation. While RDMA-capable network has a great potential to provide sufficiently high bandwidth, its current use over TCP/IP or tied to particular programming models, such as MPI, limits its capability to break the bandwidth bottleneck.**

In this work we propose iRDMA, an RDMA-based parameter server architecture optimized for high-performance network environment supporting both GPU- and CPU-based training. It utilizes native asynchronous RDMA verbs to achieve network line speed while minimizing the communication processing cost on both worker and parameter-server sides. Furthermore, iRDMA exposes the parameter server system as a POSIX-compatible file API for convenient support of load balance and fault tolerance as well as its easy use. We have implemented iRDMA at IBM's deep learning platform. Experiment results show that our design can help deep learning applications, including image recognition and language classification, to achieve near-linear improvement on convergence speed and training accuracy acceleration by using distributed computing resources. From the system perspective, iRDMA can efficiently utilize about 95% network bandwidth of fast networks to synchronize models among distributed training processes.

***Index Terms*—RDMA; deep learning; network;**

I. Introduction

Deep learning is reshaping the landscape of many traditional machine learning fields, including image classification, language processing, and machine translation. In pursuit of fast training performance, researchers have resorted to massive-parallel GPU devices to conduct deep neural network (DNN) training on a single node over the past few years [7], [16], [14], [4]. However, as the demand on better training services continues to grow, seeking of high-performance deep learning has led to the need of distributing the training workloads over multiple GPUs on a cluster of nodes. A common practice in such distributed GPU training is to rely on the assumption that each subsets of input data are independently and identically distributed (IID), and a centralized parameter server system can be used to aggregate data about learning progress from different learners.

In order to fully utilize the computing capability of all the available streaming multiprocessors within a GPU device, current deep learning frameworks usually use batch-based processing to transform a group of input data, *e.g.,* images, into a large matrix, so that efficient matrix operations from GPU libraries, such as cuBLAS and cuDNN [5], can be applied to accelerate the forward and backward passes for each individual datum. During the training, forward pass is to assess the quality of the DNN model and back propagation is to generate the gradients with respect to the current model weights used by the neural network. For each batch processing, gradients generated by different rounds of back propagation are accumulated and normalized, and then used to refine the model weights. A straightforward approach to leveraging distributed GPUs for acceleration is to split each batch into multiple mini-batches and assign different mini-batches to different GPU learners for processing. Assuming there exists a centralized parameter server to aggregate the gradients, at the end of a mini-batch processing each learner uploads its own gradients to the parameter server, which then normalizes the gradients collected from all the learners and sends back updated model weights.

However, as GPU devices become increasingly fast, network bandwidth rises as a severe bottleneck keeping distributed deep learning from achieving strong scalability. One approach to alleviate the network bottleneck is to increase the mini-batch size for lower communication frequency and longer computation time between batches. Unfortunately, such an approach can compromise training quality with a reduced convergence speed, as has been observed in previous studies [23], [12]. An alternative method is to overlap communication overhead with the computation time. However, without efficient system design the communication of data about a trained model between the learners and parameter servers, or model exchanging, can consistently lag behind the GPU computation, which leaves the learners no choice but to use staled model weights in its computation. Consequently, it may slow down the training convergence.

To address the issues, many deep learning service vendors recently pay great attention on high-performance networks, such as 56/100 Gbps InfiniBand and Converged Ethernet, to support efficient distributed GPU-based training. However, existing software solutions, such as the one used by the GeePS [9], that rely on TCP/IP-based communication mechanism are unable to fully exploit the benefits of the fast networks. In addition, although several RDMA-enabled MPI primitives, including AllReduce, can be employed to implement the gradients aggregation, they require rigid syn-

978-1-5386-2588-0/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCC-SmartCity-DSS.2017.30

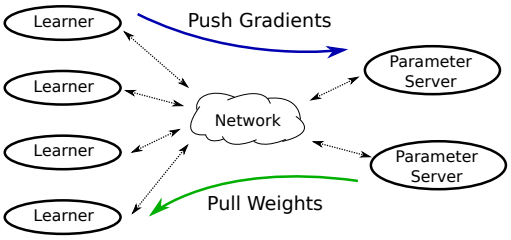


Fig. 1: Parameter server architecture.

chronization policies, which can lead to frequent GPU stalls. Moreover, lack of fault tolerance support renders MPI-based approach less attractive.

In this work, we aim to leverage Remote Direct Memory Access (RDMA) to design a high-performance model exchanging protocol, named as iRDMA, between GPU-based learners and parameter servers to accelerate distributed GPU-based deep learning systems on both throughput and convergence speed. More specifically, it takes advantage of the native asynchronous RDMA verbs to achieve network line speed for model exchanging while minimizing the communication cost on both learner and parameter-server sides. It pipelines the model transferring over the network with the gradient aggregation on the servers by dividing the gigantic neural network model into a large number of small blocks as working units. Combining this data chunking with double buffering, iRDMA can overlap the communication with both the training on the learner side and the aggregation on the servers, and evenly distribute the network traffic and aggregation workloads among multiple parameter servers. Furthermore, such a design can effectively reduce memory demand at the server side. Built atop this protocol, we further introduce an adaptive pulling algorithm that takes account of the computation and communication timing to carry out the model pulling to address model staleness issue.

There are several challenges in designing the RDMA-based model exchanging solution. First, in order to achieve zero-copy and hide communication overhead, it requires a holistic design on the network protocol and asynchronous scheduling. Second, as the default communication unit, which is the entire neural network model, can contain hundreds of millions or even billions of parameters, it is hard to determine how to perform data chunking to overlap the communication and computation. Lastly, deep learning frameworks are written in different languages, *e.g.,* C++ in Caffe [14], Python in Theano [4], and Lua in Torch [8]. It requires substantial engineering efforts to wrap the backend communication system. In order to efficiently implement iRDMA, this work has adopted a novel approach by leveraging the well-optimized high-performance components found in distributed storage systems, including the Linux SCSI framework, to assemble a high-performance parameter server system. It exposes a set of POSIX-compatible APIs to simplify the integration with different deep learning frameworks.

II. Related Work

The parameter-server-based approach is a primary method to scale out deep learning tasks in a distributed environment. DistBelief [11], Project Adam [6], and the Parameter Server [17] are three pioneering works on using parameter server systems to facilitate large-scale CPU-based machine learning and deep learning applications. The design of parameter server system commonly follows the architecture as depicted in Figure 1, which consists of a group of learners, each of which processes a part of the input training data. Periodically, learners upload local gradients or weights generated from previous training to a group of parameter servers, which aggregate the gradients received from different learners and update the global weights. The parameter servers then return updated model weights back to the learners. While several parallel algorithms [19], [22], [13], [24] have been proposed over the past few years to accelerate the distributed training, one of the major driving algorithms that enables such parallel training is Parallel Stochastic Gradient Descent [11].

However, large-scale CPU-based training is challenged by cost efficiency issue. Recent deep learning research [7], [9], [16] has demonstrated that comparable accuracy and significantly reduced training time can be received with a small number of commodity GPUs in a highly cost-efficient manner. To efficiently utilize local GPUs for training deep neural networks, many frameworks, including Caffe [14], Torch [8], and Theano [4] have been introduced over the past few years. However network bandwidth remains as a severe bottleneck preventing these systems from leveraging distributed GPUs. Currently, all the above three systems are still based on local GPU. There are several attempts to scale out distributed GPU training, including DeepSpeech [2] and GeePS [9]. However, the former takes advantages of an HPC environment to battle against the bandwidth issue while the later relaxes the batch size limit at the cost of degrading the peak accuracy.

III. Motivation

To understand the network bandwidth requirement on sustained overlapping of the DNN training with model exchanging between parameter servers and learners, we have examined three neural network models, including Natural Language Classification (NLC), VGG Convolutional Neural Network (VGG), and AlexNet. Figure 2 illustrates the required network bandwidth in a single direction to keep pace with different computing devices. As shown in the figure, when neural networks are trained on CPUs with OpenBLAS acceleration, the pressure on the network is fairly moderate as the entire training is constrained by the slow computation. However, migrating the computation from CPU to GPU (NVIDIA K40 with 1.43 TFLOPs) shifts the bottleneck from the computation devices to the the network bandwidth. In particular, a 10 Gbps Ethernet can no longer offer sufficient bandwidth to allow different learners to retrieve the latest models from the parameter servers after each mini-batch processing. To address this issue, faster networks, such as 40/56 Gbps InfiniBand, become necessary to mitigate the bandwidth bottleneck. However, such

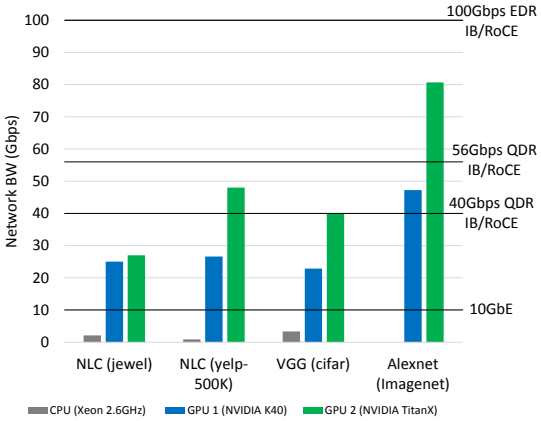


Fig. 2: Network throughput requirement for completely overlapping computation with communication for deep learning.

a solution is only temporary as faster GPU devices, such as NVIDIA Titan X with 11 TFLOPs, have significantly reduced the computation time, exposing the network bandwidth as a growing concern. Also shown in the figure, the full network bandwidth is generally needed when faster GPU device is used to carry out the training. Although TCP-based transfer can improve its performance, it requires many application and system optimizations including multi-stream-based application protocol design, receiving side steering, interrupt binding, and enlarge MTU. It also has high protocol processing cost. For example, a 10 Gbps networking saturates a CPU core on both send and receive side, then a 56 Gbps one would consume more then five cores on each side. This compromises CPU-based training due to contention on CPU cores and memory bandwidth [18], [20]. Therefore, TCP-based solution is not optimal in a fast network environment at 56 Gbps, 100 Gbps, or even higher, and it becomes inevitable to explore efficient use of RDMA networks for communication-intensive workloads such as distributed deep learning.

IV. Design and Implementation

iRDMA aims to provide a low overhead and high bandwidth data path between parameter server and a (large) number of parallel training nodes (as well as jobs running on them) for synchronizing training updates. To maximize computation resource utilization, we introduce a zero-stall design by using a double-buffer mechanism. Meanwhile, iRDMA introduces an adaptive pulling mechanism to minimize staleness between workers on the training nodes and the global model on the parameter server. From the system performance perspective, iRDMA utilizes a block based data transfer method, in which a large model is split into small pieces to maximize transmission throughput and reduce memory demand at the parameter server. iRDMA also provides a POSIX-compatible programming API to ease the integration efforts as well as reduce the complexity on model sharding and fault tolerance.

A. Optimized RDMA data path for distributed deep learning

RDMA and its supporting network layer, such as InfiniBand and RoCE (RDMA over Converged Ethernet), are becoming a dominant cross-node data exchanging path to support communication-intensive workloads for their high bandwidth at 40/56/100 Gbps and low latency at single-digit microsecond scale. However, it often demands to take consideration of workloads characteristics for a co-design of application network protocol and user-space memory management to fully exploit advanced RDMA performance advantage. The distributed deep learning workloads generate high frequent requests of data into and out of parameter server periodically. The I/O size mostly is in the megabyte to gigabyte scale, and I/O throughput is the key performance metric. Therefore, how to fully utilize the available network bandwidth becomes critical in achieving high performance.

RDMA offers a set of of design choices for a variety of workloads with different characteristics. There are two message transfer semantics: one-sided and two-sided channel semantics. Because deep learning workload transfers large messages, the two-sided channel has a higher memory demand on the parameter server side. In contrast, one-sided operations, including RDMA write and RDMA read, are optimized for throughput-oriented workloads. They bypass the receiving side notification stack, thus demand an out-of-band notification mechanism. The one-sided operations can share a memory pool and dynamically allocate different memory region accordingly. Accordingly, iRDMA uses one-sided RDMA operations to achieve high throughput.

To further improve the throughput of aggregation functions, iRDMA maintains model data inside the memory of the GPU device on the parameter server, and transfers *gradients* into and *weights* out of GPU memory directly by using GPUDirect RDMA technology. For the `write()` operation, the latency consists of network transmission and model aggregation. The GPUDirect RDMA has similar network transmission performance as regular main memory based RDMA operations, while the massive GPU parallelism decreases the aggregation time. On serving a `read()` operation, however, GPUDirect RDMA incurs the PCIe host interface overhead on in-flight transactions from a GPU device. As a result, the I/O requests can only achieve half of the network peak throughput, and the latency for retrieving model from GPU's memory is 2× higher than that from server's main memory. This offsets the benefit iRDMA obtains from the write() operation. To address the issue, iRDMA creates a dedicated GPU stream to move updated weights to the main memory. The `read()` request would retrieve updated weights from the host memory, as shown in Figure 4a. While the upcoming PCIe peer-to-peer engine is expected to improve `read()` throughput, iRDMA would leverage the GPUDirect RDMA to provide even lower end-to-end latency to the workers.

B. Model buffer switching and adaptive pulling

A training iteration takes a small batch of training sample to compute the gradients according to the neural network model

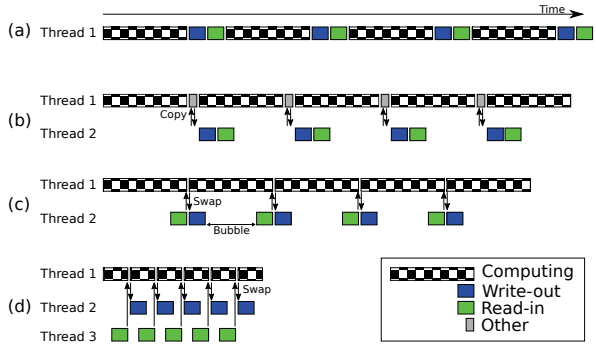
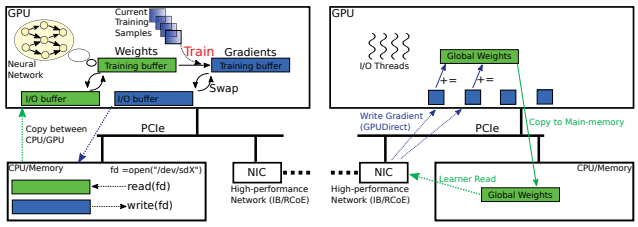


Fig. 3: Data-exchange schemes for distributed training.



(a) Learner side design. (b) Parameter Server design.

Fig. 4: iRDMA design on learner side and on parameter server side.

definition and current model weights. In a distributed environment, each worker contributes the gradients to the global weights, and requests for the most updated weights to start the next iteration. Synchronized communication may stall the GPU and CPU computation during the communication phase. Meanwhile, the network becomes idle during computation phase, as shown in Figure 3(a).

Asynchronous communication scheme overlaps the training and the communication phases to improve resource utilization. There are two issues. First, the gradients copy-out and weights copy-in between model memory and GPU communication buffer take notable time, especially with large models of hundreds of megabytes or even several gigabytes. Second, the prefetched data is often of a version copy of the global weights right after the gradient is sent to the parameter server. There is a staleness time window on the worker side after the global weight is downloaded and before the next iteration starts.

We propose a buffer switching mechanism to address the first issue. iRDMA separates the model data from model definition (metadata) by allocating two buffers for holding the model data. At the beginning of an iteration, the worker uses one of of model data buffers (a weights buffer and a gradients buffer) to store and access its model definition, as shown in Figure 4a. After the computation completes, the worker sends a notification to the backend communication thread to switch the buffer for the model definition to the other model buffer. During the gradients computation, the backend communication thread sends the last iteration's gradient contribution, and pulls the latest weights back. This avoids intermediate data copy between the computation thread and communication thread, thus enables near-optimal computation resource utilization.

In an optimal case, data pulling and computation finish at the same time in each iteration. On the next iteration, the worker can use the most updated weights with minimum staleness. To address the second issue, we design an adaptive pulling mechanism to pull the global weights. The communication thread collects a number of recent computation interval time periods and model-downloading latencies to calculate their moving averages for predicting the completion time of current iteration, and then inserts a sleep time, named *bubble*, adaptively, as depicted in Figure 3(c). In some rare cases, the computation time is shorter than the latency of writing-out gradients and read-in weights. The worker uses two threads to overlap the bidirectional data transfer, as shown in Figure 3(d).

C. Block-based communication design and optimization

The model updates (*gradients*) and the model data (*weights*) are the transfer entities between the workers and the parameter server in each iteration. Their sizes are often in the range of a few megabytes to a few gigabytes containing millions to billions of floating point numbers. Traditionally, the whole entities are transferred in an all-in-one-piece manner to simplify the design and implementation of memory management and RDMA-based application communication protocol. For example, allocating a specified memory area for each worker avoids dynamic memory management, and the parameter server aggregates the updates only after the whole model entity arrives. However, it incurs high memory demand as the parameter server needs to allocate a memory buffer as large as the product of model size and number of workers for each training job. This would consume tens of gigabytes main memory. Also, there is only one request that can be merged at a given time since the merging cannot be performed in parallel. As a result, some of the workers may expect stall on the communication.

To achieve the bare-metal bandwidth of RDMA networks, iRDMA uses a block-based communication mechanism. It divides the weights and gradients entities into sub-megabyte blocks, each of which is associated with a unique block ID. The parameter server maintains the global model status in blocks while the workers send gradients and request for updated weights with block IDs. iRDMA creates a group of threads to service concurrent requests in parallel. The aggregation latency on each block can be reduced due to much reduced data volume in each transfer (compared to the whole model aggregation method). In the meantime, the I/O parallelism is increased among different workers. Furthermore, to transfer the same amount of data, RDMA network interfaces (RNIC) can achieve better throughput with a set of I/O requests in the kilobyte range rather than with one request requiring a large memory buffer [21].

Because block-based communication between workers and the parameter server has similar I/O data path as existing network block storage systems, we build iRDMA by leveraging the Linux SCSI target framework, `tgt`, which supports both

TCP/IP and RDMA networks [10]. Existing storage servers keep the contents in persistent storage media, and usually map a virtual storage address, *logical block address*, in the I/O request to a physical storage address. In the scenario of communication in the deep learning platform, each I/O block represents a part of the global model data, and workers send write requests to the parameter server. The parameter server needs to merge the gradients into corresponding weights range. We implement an aggregation function for each of the I/O blocks, and register it as a callback function on serving the write requests. The global model data is maintained in the main memory of parameter server for fast aggregation and model retrieving.

D. Scalability, fault-tolerance, and interoperability

Scalability. To serve large-scale training jobs employing ten or even hundreds of GPUs or CPUs, the parameter server needs to scale out by utilizing computation (for aggregation) and networking (for I/O) resources located on different machines. iRDMA splits the global model data into partitions, and each partition is served by a parameter server shard processing block-based requests. Each of the worker nodes initiates the parameter server shards as a group of block devices. So the workloads can be evenly distributed to the backend parameter server shards. Meanwhile, the address translation and model data partition are hidden to the user and developer, and can be done through configuration to simplify client library design and implementation.

Fault tolerance. iRDMA can tolerate both learner failure and parameter server failure. When a worker dies unexpectedly, iRDMA continues to aggregate gradients pushed from remaining learners. The learners then repartition the training data and keep refining the training model. Only when the number of dead workers reaches a threshold does iRDMA terminate the training job.

Interoperability. To enable distributed training, the deep learning community often either use distributed networking programming and integration or adopt distributed programming models and libraries such as MPI. This requires substantial integration efforts from the deep learning experts, and different frameworks written in different programming languages require an additional wrapper to interoperate with the parameter server. To tackle the problem, iRDMA abstracts the distributed global training model as a block device such that the workers can access it directly through POSIX file I/O APIs, i.e. `write` and `read`. The communication channel setup and address space mapping can be performed through configuration instead of programming. Because most of the programming language has built-in file I/O library, those frameworks can operate on iRDMA directly. This significantly reduces the development effort.

V. Evaluation

We have evaluated iRDMA extensively with both micro-benchmarks and real-world workloads. Below we will present performance results of iRDMA in different network environments in comparison with TCP alternatives. We will also examine the end-to-end acceleration by enabling distributed training in natural language classifications and large-scale image recognition applications.

A. Experimental setup

Our testbed consists of three servers with GPU and RDMA supports. Each server is equipped with two NVIDIA Tesla K40m GPUs. All of the nodes are connected through both 10 Gbps Ethernet and Mellanox 56 Gbps FDR InfiniBand. To enable GPUDirect RDMA, we installed the Mellanox peer-to-peer driver for NVIDIA GPUs. We set up two of the server as the worker nodes and the third one as the parameter server. The aggregated computation power of four GPUs for training reaches 5.6 TFLOPs. The InfiniBand bandwidth is limited by PCI express 8-lane throughput and InfiniBand header overhead, and in theory the peak payload throughput can reach 50 Gbps.

B. Microbenchmark

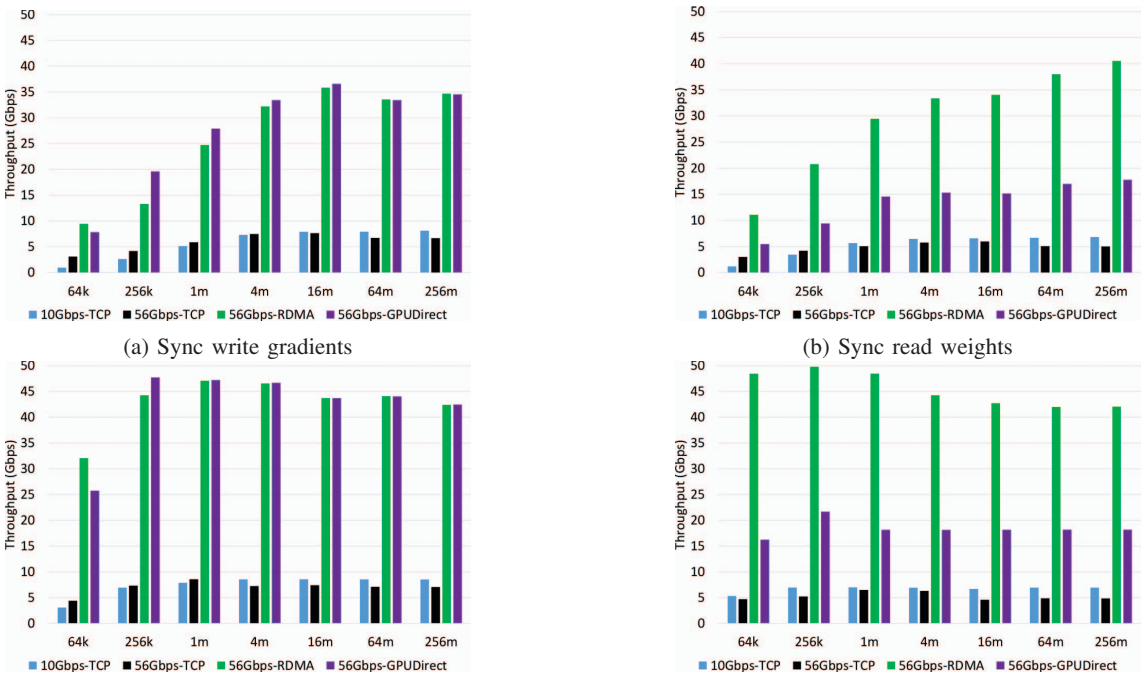
During training, the parameter server process performs communication and aggregation. The most critical performance metric for iRDMA is the aggregated processing throughput under massive concurrent read and write requests from worker nodes. We perform a set of experiments with micro-benchmark to evaluate iRDMA server performance.

We have developed a micro-benchmark derived from fio [3], a popular file I/O stress tool. As described in section III, iRDMA provides a block device abstraction to the worker nodes, and the worker processes can write gradients and read updated weights through the network block device. So the microbenchmark generates the same amount of data as that of the neural network models to simulate the gradients generated by learners, and writes them to the parameter server. iRDMA then performs aggregation function upon receiving the gradients, and returns the aggregated weights to the clients per read request.

The block device I/O can be performed in either a synchronous way, i.e. `read/write` or an asynchronous way with Linux `libaio` (named as AIO in the below). The AIO can split a relatively large I/O request into many smaller I/O requests and send the requests in parallel to I/O device. As a result, AIO can queue a list of I/O requests to the I/O device to achieve higher performance. In the iRDMA context, the AIO can also be used to split the gradients buffer into smaller pieces and send them in parallel.

We have measured throughput of parameter server by deploying iRDMA using TCP, RDMA, or GPUDirect RDMA as the transmission methods. Figure 5 shows the training throughput under different I/O patterns, in which gradients and updated weights are partitioned into data chunks of different I/O sizes and transferred between learners and servers via either synchronous or asynchronous methods.

In the evaluation we have a number of interesting observations. First, the RDMA-based solution outperforms the TCP-based solution in all of the test cases, and the TCP data path



(a) Sync write gradients

(b) Sync read weights

(c) Async write gradients

(d) Async read weights

Fig. 5: System throughput with synchronous and asynchronous reads and writes.

cannot scale up with high-performance networks. In particular, the 56 Gbps-TCP setup requires a software network header translation service, IP over IB (IPoIB), which significantly limits the TCP bandwidth in the fat link and often bounded by a CPU processing capability. Second, asynchronous I/O performs better than synchronous I/O. Because there is at most one I/O request in flight in the synchronous I/O, network delay cannot be overlapped with batched requests. As a result, synchronous I/O can only utilize about 80% of the available network bandwidth. In contrast, AIO can achieve more than 95% network bandwidth with the block size in the range from 256 KB to 4 MB, which is a sweet spot for iRDMA. Smaller requests may incur excessive control event-processing cost while larger requests incur NIC context control overhead. Third, GPUDirect RDMA achieves comparable performance for write operations, but only achieves about 50% network utilization for read requests. For example, the 256KB AIO write with GPUDirect RDMA achieves the best write performance, while the 256KB AIO read with GPUDirect only receives less than half of the available bandwidth. Therefore, iRDMA uses GPUDirect RDMA to serve write requests and regular RDMA for read requests.

The reason why the end-to-end GPUDirect RDMA cannot efficiently saturate the network bandwidth and performs much worse than RDMA protocol is that PCIe controller restricts the number of transactions from GPU to other PCIe-connected devices. As a result, RDMA-based NICs can only utilize half of the PCIe peak bandwidth for moving data out of GPU memory directly.

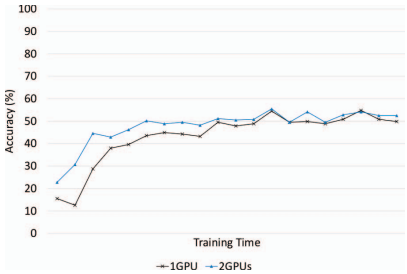


Fig. 6: Language classification accuracy over time.

C. Real-world workloads

In this section, we further evaluate iRDMA with real-world deep learning workloads using natural language classification and image recognition. We show the performance improvements achieved by our design in three metrics, namely convergence speed, accuracy, and number of images processed per second.

1) Natural language classification: Natural language classification represents a class of text classification workloads. Our training sample includes 2,400 sentences split into 311 categories. Normally, its training samples are not in a big search space. Therefore, the training converges relatively fast with a few iterations. We use two GPUs on different servers to train the model in parallel and exchange updates using iRDMA. Figure 6 shows the accuracy improvement with respect to time. iRDMA provides a higher converge speed by fully overlapping communication with computation, and two

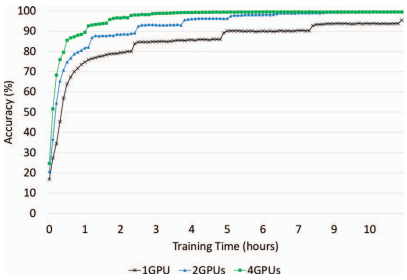


Fig. 7: Cifar-10 training accuracy

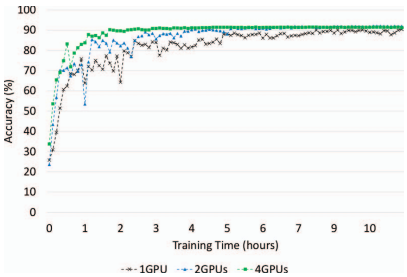


Fig. 8: Cifar-10 test accuracy

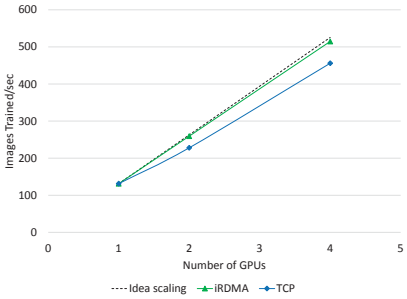


Fig. 9: Cifar-10 GPU scalability

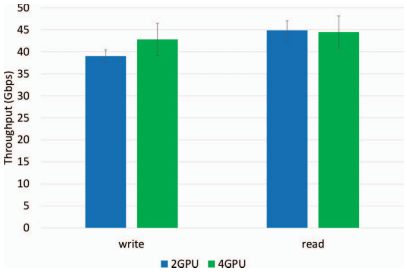


Fig. 10: Parameter server network bandwidth on serving write gradients and read weights operations

GPUs can train 2× training samples than a single GPU with the same amount of time.

2) Cifar-10 image recognition: The Cifar-10 data set consists of 60,000 32x32 color images labeled in 10 mutually exclusive classes [15]. The data set is split into 50,000 training images and 10,000 test images. We have integrated Torch with iRDMA to enable distributed training, and used a Torch-based application [1] to train a model to classify images. We chose VGG (Visual Geometry Group) neural network model consisting of 1.5 million model parameters.

We measured the training performance in terms of convergence speed and quantified training progress based on the training and test accuracy achieved over time. Figures 7 and 8 show the training accuracy and test accuracy improvement, respectively, over the training time. We run local GPU training without communication overhead, shown as 1 GPU in the figures, and scale out the training with iRDMA by using two and four GPUs across two worker servers. The two worker servers communicate updates by using the iRDMA over RDMA, shown as 2 GPUs and 4 GPUs.

As shown in the figures, the distributed GPU training using iRDMA delivers a higher convergence speed compared to single GPU training. For example, to get 90% training accuracy, a single GPU spends 5.3 hours, while it takes 2.5 and 1.3 hours for 2 GPUs and 4 GPUs, respectively, to achieve the same accuracy. This suggests strong scalability on training performance. Second, multiple GPUs are able to train more samples in the same amount of time compared to a single GPU, demonstrating that iRDMA is able to aggregate the updates from multiple learners and broadcast the weights efficiently. Figure 9 shows the training throughput in terms of trained images per second. iRDMA achieves near-linear training scalability with the double buffering design and efficient RDMA communication, while TCP-based solution incurs notable communication overhead and cannot overlap communication with computation. In addition, Figure 10 shows the iRDMA's average bandwidth on serving read and write requests. Because serving write operations includes aggregation time as well as communication time, the overall bandwidth on serving write requests is smaller than that for read requests. The read operations can achieve 95% network utilization on average.

VI. Conclusion

Network bandwidth is rising as a major bottleneck keeping GPU-based deep learning from scaling out. To address this issue, we introduce iRDMA, an RDMA-accelerated parameter server system to allow existing deep learning frameworks to exploit the full bandwidth of fast networks. By optimizing the model exchange channels through combining the native asynchronous RDMA verbs with efficient model partitioning and using double-buffer mechanism, iRDMA delivers near-line-rate model exchanging rate with minimal stalls on GPU computation due to fully overlapped computation and communication. In addition, adaptive pulling mechanism effectively reduces the model staleness and increases the learning efficiency of all the learners. Moreover, iRDMA is designed via a novel usage of conventional network storage system and exposes POSIX-compatible API to access shared global model, which effectively reduces the engineering efforts to support load balance and fault tolerance, and simplifies the integration with existing deep learning frameworks.

VII. Acknowledgments

We are grateful to the paper's reviewers who helped to improve the paper's quality. This work was patially supported by NSF of China under Grants No.61572487 and No.61472323.

References

[1] "Cifar.torch." [Online]. Available: https://github.com/szagoruyko/cifar.torch

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015. [Online]. Available: http://arxiv.org/abs/1512.02595

[3] J. Axboe, "Flexible I/O Tester," 2016. [Online]. Available: http://freecode.com/projects/fio

[4] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[5] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 571–582. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[7] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th ICML*, 2013, pp. 1337–1345.

[8] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[9] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 4:1–4:16. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901323

[10] D. Dalessandro, A. Devulapalli, and P. Wyckoff, "iSER storage target for object-based storage devices," in *Proceedings of Fourth International Workshop on Storage Netwo rk Architecture and Parallel I/Os*, September 2007.

[11] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., 2012, pp. 1232–1240. [Online]. Available: http://books.nips.cc/papers/files/nips25/NIPS2012_0598.pdf

[12] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning," *IEEE International Conference on Data Mining 2016*, 2016.

[13] 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 26*, 2013, pp. 1223–1231.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[15] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

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

[17] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[18] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:IJPP.0000029272.69895.c1

[19] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 693–701.

[20] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi, "Design and performance evaluation of numa-aware rdma-based end-to-end data transfer systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 48:1–48:10. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503260

[21] Y. Ren, T. Li, D. Yu, S. Jin, T. Robertazzi, B. L. Tierney, and E. Pouyoul, "Protocols for wide-area data-intensive applications: Design and performance issues," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 34:1–34:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389043

[22] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD," *CoRR*, vol. abs/1412.6651, 2014.

[23] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, pp. 2350–2356.

[24] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems 23*. Curran Associates, Inc., 2010, pp. 2595–2603. [Online]. Available: http://papers.nips.cc/paper/4006-parallelized-stochastic-gradient-descent.pdf