Fast Distributed Deep Learning over RDMA

Jilong Xue
Microsoft Research
jxue@microsoft.com

Youshan Miao
Microsoft Research
yomia@microsoft.com

Cheng Chen
Microsoft Research
cncng@microsoft.com

Ming Wu
Microsoft Research
miw@microsoft.com

Lintao Zhang
Microsoft Research
lintaoz@microsoft.com

Lidong Zhou
Microsoft Research
lidongz@microsoft.com

Abstract

Deep learning emerges as an important new resource-intensive workload and has been successfully applied in computer vision, speech, natural language processing, and so on. Distributed deep learning is becoming a necessity to cope with growing data and model sizes. Its computation is typically characterized by a simple tensor data abstraction to model multi-dimensional matrices, a data-flow graph to model computation, and iterative executions with relatively frequent synchronizations, thereby making it substantially different from Map/Reduce style distributed big data computation.

RPC, commonly used as the communication primitive, has been adopted by popular deep learning frameworks such as TensorFlow, which uses gRPC. We show that RPC is sub-optimal for distributed deep learning computation, especially on an RDMA-capable network. The tensor abstraction and data-flow graph, coupled with an RDMA network, offers the opportunity to reduce the unnecessary overhead (e.g., memory copy) without sacrificing programmability and generality. In particular, from a data access point of view, a remote machine is abstracted just as a "device" on an RDMA channel, with a simple memory interface for allocating, reading, and writing memory regions. Our graph analyzer looks at both the data flow graph and the tensors to optimize memory allocation and remote data access using this interface. The result is up to 169% improvement against an RPC implementation optimized for RDMA, leading to faster convergence in the training process.

CCS Concepts • **Software and its engineering → Distributed systems organizing principles**; • **Computing methodologies** → *Neural networks.*

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303975

ACM Reference Format:
Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast Distributed Deep Learning over RDMA. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3302424.3303975

1 Introduction

Deep learning, in the form of deep neural networks (DNN), is gaining popularity thanks to its huge success in areas such as speech, vision, and natural language processing. There is a trend of using deeper, more complex neural network models trained with increasingly larger data sets. Such a model often takes hours, days, or even weeks to train on a CPU/GPU cluster. The deep learning computation in training a model involves multiple iterations with rather frequent synchronizations. The performance therefore often critically depends on the efficiency of cross-machine communication, including its ability to leverage emerging network technology, such as Remote Direct Memory Access (RDMA).

Remote Procedure Call (RPC) is a widely used general-purpose communication paradigm. In addition to data transfer, RPC takes care of data serialization and deserialization for various data types, manages communication buffers, and handles message assembly and batching automatically. Even with RDMA, RPC can be used to help mediate concurrent (remote) writes to the same data [21]. It is therefore natural for deep learning frameworks such as TensorFlow [7] to adopt gRPC, a form of RPC, as its communication abstraction.

In this paper, we argue against using RPC for distributed deep learning computation, especially on an RDMA-capable network. This is because (i) deep learning computation uses tensor (or multi-dimensional matrix) as the main data type, which consists of a plain byte array as tensor data and a simple schema as meta-data specifying the shape and element type of the tensor. A tensor is often of a sufficiently large size (tens of KB to MB) and its metadata/data sizes often static. Using RPC for tensor data transfer does not provide evident advantage on programmability or efficiency; and (ii) using RPC typically involves memory copy to and from RPC-managed communication buffers. Zero-copy cross-machine tensor transfer is possible with RDMA because the source and destination tensors can be appropriately allocated in the

RDMA memory region and known statically. We therefore advocate a simple and almost trivial interface that exposes a remote machine as a "device" from a data access point of view. This "device" is connected through an RDMA-based channel that exposes control for parallelism. Remote memory regions can be allocated and directly accessed through this "device" interface, much like a local GPU. This maps naturally to the underlying RDMA network that provides direct remote memory access. It is worth pointing out that previous work on efficient communication on RDMA often uses RPC (e.g., for writes) partly because they are focusing on variable (and often small) size data transfer for key/value stores, where they can benefit from batching in RPC and from mediating concurrent remote writes to the same region through RPC [16, 19, 20, 26]. Neither is necessary in our case.

We have designed a zero-copy cross-machine tensor transfer mechanism directly on our "device" interface. This is done through a combination of static analysis and dynamic tracing on the data-flow graph of the computation in order to (i) figure out whether the size of each tensor that needs to be transferred across server can be statically known at the compile time, (ii) assess whether such a tensor should be allocated statically (for better efficiency) or dynamically (for reduced memory footprint), (iii) ensure allocation of the tensors on both the sending and receiving ends in the RDMA memory regions, (iv) identify the source and destination addresses of tensors for RDMA-based transfer.

We have implemented an efficient RDMA-based "device" library, and integrated it with our graph analysis and tracing mechanism into the data-flow graph runtime of TensorFlow [7] for tensor data transfer in distributed deep learning computation. The experiments show that our proposed techniques help TensorFlow achieve up to 169% improvement in representative deep learning benchmarks against an RPC implementation optimized for RDMA.

2 Background and Problems

In this section, we introduce the relevant background about data-flow graphs in deep learning, RPC, and RDMA, leading to the potential issues of distributed deep learning over RDMA with RPC.

2.1 Deep Learning Data-Flow Graph

Deep neural network describes a layered machine learning model that consists of neurons connected through synapses. The layered structure enables the model to learn hierarchical features from the raw input data. Each layer normally represents a linear transformation on its inputs followed by some non-linear activations. The left side of Figure 1 shows an example of a vanilla deep neural network. The parameters to learn in this model are the weights of the connections between neurons of different layers. The computation on this model can be naturally expressed using a data-flow graph

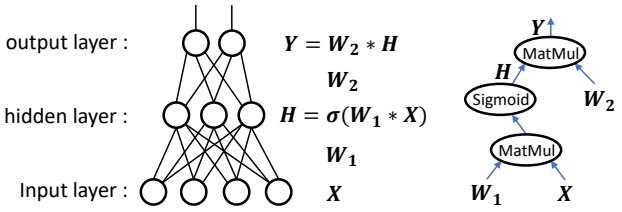


Figure 1. Example of a vanilla neural network (left) and the data-flow graph (right) of its forward computation. σ is the non-linear *Sigmoid* function. Bold symbols are the variables representing tensors.

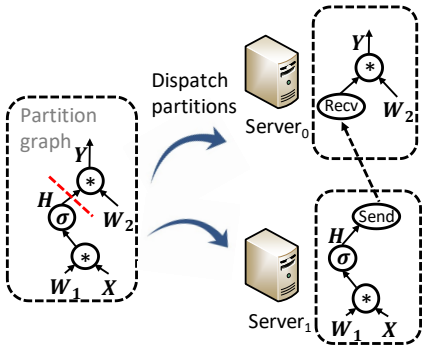


Figure 2. Overview of the distributed architecture of deep learning data-flow computation. Dotted-line arrow refers to the cross-server data flow.

where the nodes represent the computations at layers and the edges represent the data flowing between the dependent nodes. In deep learning scenarios, the major data type flowing in the graph are tensors (i.e., multi-dimensional matrices) because most deep learning algorithms are expressed as mathematical models on matrices. The right side of Figure 1 shows an example data-flow graph expressing a forward computation on the neural network in the figure from the raw input to upper layers. Through supporting this data-flow graph representation for deep learning computation, frameworks [7, 9, 14, 33, 40] can allow developers to conveniently implement variant forms of neural networks that can be complex.

The training process of a deep neural network can be time-consuming because the computation needed to learn complex hierarchical features is complex and often involves processing large volumes of training data. In order to scale out the computation, distributed deep learning can be applied by replicating and partitioning the data-flow graph onto multiple servers to execute in a data-parallel or model-parallel fashion. As illustrated in Figure 2, a data-flow graph can be partitioned with each partition placed on a different server before the computation is conducted. The data flow between graph nodes across partitions will be fulfilled through the underlying communication layer during the computation.

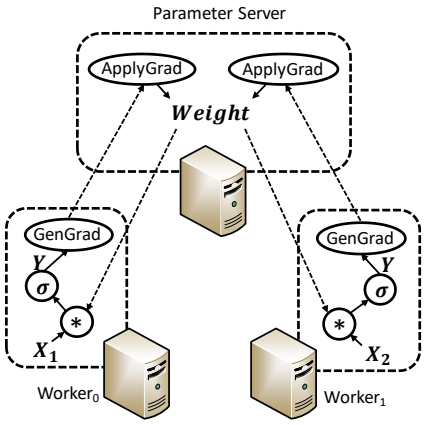


Figure 3. Example of distributed data-flow computation of deep learning with data-parallelism in a parameter-server architecture. Dotted-line arrows correspond to cross-server data flows. For simplicity and clarity, *GenGrad* and *ApplyGrad* represent the sub-graphs of computing and applying gradients, respectively. *Weight* is the tensor representing the model parameters, which is shared by all workers.

Figure 3 shows an example of distributed deep learning computation with data-parallelism. A data-flow graph is replicated on two workers and each replica is partitioned among a worker and a parameter server. Employing such a distributed data-flow graph model offers convenience and flexibility to allow not only data-parallelism, but also model-parallelism, which is critical when the deep learning model size is large.

Deep learning training involves iterative executions over the data-flow graph for multiple mini-batches of training data. Therefore, after the data-flow graph is created and before the computation starts, it is reasonable for a deep learning framework to take some time to analyze the graph and optimize the execution. In the graph analysis phase, a static graph-analysis phase can extract useful high-level information to be passed to a lower layer and used to improve runtime efficiency. One example of such information is the addresses of the tensor data that need to be transferred across servers. This information can be obtained statically sometimes, because the shapes of some tensors do not change during the entire computation; e.g., in the case of the tensors representing the model parameters. The framework can then arrange the placement of those tensors in memory before the execution of the data-flow graph. It is therefore feasible to design an appropriate abstraction for the communication layer to accept such information to improve its efficiency.

2.2 Remote Procedure Call

Remote Procedure Call (RPC) [8] is a common abstraction for communication across servers. It allows users to implement a procedure that can be invoked remotely as if being called locally. With RPC, users only need to focus on the implementation of the functional logic of the remote procedure without caring about the underlying communication-related details. In addition, RPC is often used to pass structured messages with integrated serialization and deserialization capabilities. There are many existing designs and (open-sourced) implementations of RPC [2, 4, 6, 19] from industry and research communities. They have been extensively applied in many distributed systems [7, 9, 21, 34, 36].

The RPC abstraction is designed to facilitate the transmission of arbitrary types of messages (in any data schema or size) at any time point. This flexibility is not particularly beneficial in the deep learning scenario mainly because the major data abstraction is the *tensor*, whose meta-data contains only simple schema with shape and element type information. There is an inherent cost associated with providing this general convenience: it makes hard for the communication library to be aware, in advance, of which user buffer the received message should be directly delivered to. Therefore, a common way is to use a fixed in-library buffer to receive a message from the operating system layer and then copy the data to the appropriate user buffer. An in-library buffer is associated with each channel and should have a limited size; otherwise, there will be a scalability issue of memory consumption when the cluster of servers become large. And also, when a sender wants to transmit a message larger than the buffer on the receiver, the message has to be split into multiple fragments with each having some header information added for re-assembling at the receiver. This often require an additional data copy at the sender. The data-copy overhead is proportional to the message size, and hence can be significant when message is large. Without re-designing the abstraction, it is hard, if not impossible, to eliminate such overhead completely in the communication layer.

2.3 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is an emerging fast network technology that allows one server to directly access the memory of a remote server without involving the operating system at any endpoint. With the technology maturing and cost competitive, RDMA has found its way into data centers and is gaining popularity [26].

The user interface to issue RDMA operations is through functions called *verbs*. There are two types of verbs semantics: memory verbs and messaging verbs. The memory verbs include one-sided RDMA reads, writes, and atomic operations. These verbs specify the remote memory address to operate on without involving the remote CPU. The elimination of CPU overhead at remote side makes memory verbs attractive. The messaging verbs include the send and receive verbs, which involve the remote side CPU. Verbs are posted by applications to queues that are maintained inside the RDMA NIC. Queues always exist in pairs with a send queue and a receive queue forming a queue pair (QP). Each queue pair has an associated completion queue (CQ), which the RDMA NIC fills in upon completion of verb execution. RDMA

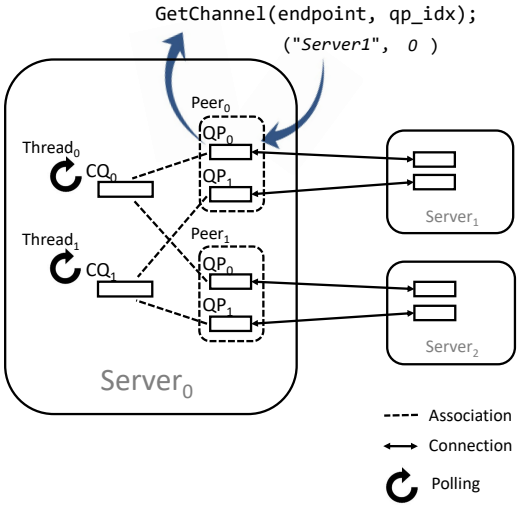


Figure 4. Overview of the architecture design of the RDMA memory copy library. QPs are created and grouped peer by peer and associated with CQs in a round-robin way.

transports can be either reliable or unreliable, and either connected or unconnected (also called datagram). In our work, we always use reliable connected transport.

RDMA networks provide high-bandwidth and low latency: NICs with 100 Gbps bandwidth and ~2μs round-trip latency are commercially available. The high-bandwidth of RDMA and its kernel-bypassing nature make any communication related computation overhead significant. We observe that removing the extra message-data copy can evidently improve communication efficiency. Simply building a general RPC abstraction over RDMA makes it hard to avoid the extra data copy. For example, the message passing mechanism used in the FaRM RPC [16] employs a fixed ring buffer with each channel on the receiver side and may suffer from the problem described in §2.2.

The one-sided memory read/write semantic of RDMA allows a zero-copy communication across servers as long as the remote address is known. For deep learning computation, the data-flow graph analysis can help arrange the in-memory placement of tensors and provide such information to the underlying communication layer. We therefore advocate exposing a simple memory-copy interface directly because tensor is the major data type to be transferred across servers during deep learning computation.

3 Design

We now present our design for the RDMA device communication abstraction, the tensor transfer mechanisms, and the integration with the deep learning data-flow graph analysis.

3.1 RDMA Device Abstraction

Our communication library provides a simple abstraction for each RDMA NIC as an RDMA device (or *device* for short

Figure 5. Transfer statically placed tensor through one-sided RDMA write.

when there is no confusion). The device provides an interface to allocate and free a memory region that can be accessed by other devices remotely. Given a remote device specified as an endpoint (i.e., IP address and port), users can acquire a channel from the local device object that connects the local device and the remote one. A channel corresponds to an RDMA QP and provides a memory copy interface for cross-server data transfer, which takes a pair of local/remote memory regions and a transfer direction as arguments. The actual data transfers are performed using the one-sided RDMA read/write verbs. To use the memory copy interface, one has to know the address of the to-be-accessed remote memory region. The library, therefore, also provides a simple vanilla RPC mechanism implemented using the RDMA send/recv verbs for this auxiliary purpose of distributing remote memory addresses. This address distribution process is often not on the critical path of the application, and hence not performance critical. Table 1 summarizes the interfaces of our communication library.

The RDMA device is configured with the number of CQs per device and the number of QPs for each connected peer remote device. The library maintains a thread pool with each thread polling a specific CQ for completion of RDMA events. When establishing a connection to a remote peer device, it evenly spreads the associations of the created QPs with the CQs in a round-robin fashion. The channel acquiring interface allows users to specify the specific QP that the channel uses. Through this interface, a multi-threaded workload (e.g., the deep learning graph execution runtime) is able to balance the loads and synchronization cost over the QPs and CQs to achieve good parallelism and communication efficiency. Figure 4 shows an overview of this design.

3.2 Transfer with Static Placement

During the graph analysis phase, the shapes of some tensors can be statically decided and will not change during the entire computation. Examples include those tensors holding the parameters of the model to be trained. Given this information, the analysis engine can allocate the memory regions for these tensors beforehand and fix their placement during the computation. If the content of such a tensor relies on that of a remote one, its address, which is remotely accessible, is distributed to the server that holds the remote upstream tensor before the computation. The sender of the tensor transfer can then use the memory copy interface to

Interfaces	Description
RdmaDev CreateRdmaDevice(num_cqs, num_qps_per_peer, local_endpoint)	Create and initialize an RDMA device associated with a local host endpoint.
MemRegion RdmaDev::AllocateMemRegion(size_in_bytes)	Allocate an RDMA-accessible memory region with specified size on a local device.
RdmaChannel RdmaDev::GetChannel(remote_endpoint, qp_idx)	Get a communication channel connecting to a remote host endpoint using a specified QP.
void RdmaChannel::Memcpy(local_addr, local_region, remote_addr, remote_region, size, direction, callback)	Asynchronously copy data between local and remote addresses with specified size and transfer direction.

Table 1. RDMA device interfaces.

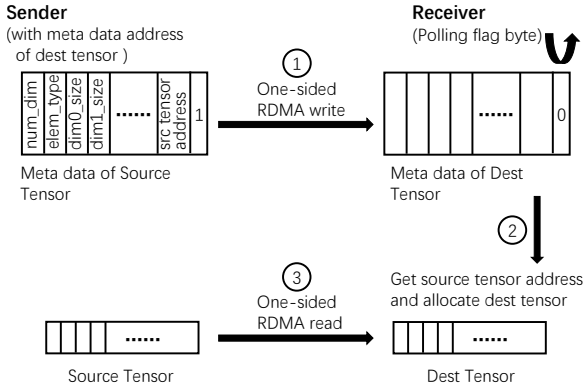


Figure 6. Transfer a dynamically allocated tensor through one-sided RDMA write and read.

write the content of the downstream tensor at the receiver directly during the computation. The receiver needs to know whether the content of the downstream tensor has been written in full. This is achieved through introducing a flag byte at the tail of the tensor memory region. The sender transfers the tensor content together with the flag byte set. The transfer is conducted in an ascending address order. The flag is then the last byte being transferred. Many RDMA NICs (including the ones we are using) guarantee that the RDMA writes are performed in an ascending address order (same as reported in FaRM [16]). So, once the flag byte is delivered, the entire tensor content must have been written in full. The receiver periodically polls the flag byte of the downstream tensor. Once the tensor transfer completes, it clears the flag for future use and then activates the graph nodes that depend on this transferred tensor for execution. Figure 5 illustrates this mechanism. Polling on a receiver has a lower priority than other ready tasks in order not to block them and to minimize its impact.

3.3 Transfer with Dynamic Allocation

It is not always the case that tensor placement can be decided statically. The shapes of the tensors to be transferred across servers can depend on the training data in each mini-batch iteration, and hence can change across different mini-batches. This is often the case where the deep learning applications have training datasets with sparse features; e.g., an RNN model for natural language processing [30] with input sequences having variable lengths in different mini-batches, or a wide-and-deep model used for recommender systems [11] with each training sample containing a different set of features.

For these cases, although the graph analysis engine cannot determine tensor placement statically, we still follow our design principle to reduce the communication-related computation overhead in a best effort. We observe that, despite variations in tensor shapes, the number of dimensions of a tensor remains unchanged throughout computation. A fixed tensor-dimension count means that the size of the meta-data of a tensor is unchanged. With this observation, we adapt the tensor transfer mechanism as illustrated in Figure 6.

As shown in the figure, the meta-data includes the number of dimensions, the size of each dimension, the element data type of the tensor, and the remote address of the tensor at the sender. The meta-data of the tensor at the receiver is preallocated and its address is distributed to the sender beforehand. During the computation, the sender writes the meta-data at the receiver through the memory copy interface when the tensor at the sender is ready to use. The receiver polls the flag byte at the tail of the meta-data. Once it detects the completion of the meta-data writing, it clears the flag byte, allocates a new tensor storage in the RDMA accessible memory region, and issues a remote memory copy to transfer the tensor data through one-sided RDMA read. Compared with the case of transferring statically placed tensor, the mechanism for passing a dynamically allocated tensor incurs the additional overhead of tensor allocation and meta-data serialization and transfer.

3.4 RDMA-Aware Graph Analysis

Given the tensor-transfer mechanisms across servers through direct memory access, the data-flow graph analyzer can be enhanced to collect and provide useful information to make communication more efficient.

Preallocate data buffers. First, for each tensor to be transferred over the network, the analyzer needs to decide whether its shape can be known statically. This can be achieved through a static shape inference process: 1) we identify the

initial set of input tensors with static shapes from the program directly as those shapes are specified explicitly through a deep learning framework (e.g., TensorFlow), and 2) we use the shape-inference functions of graph nodes to infer the set of output tensors with static shapes recursively from the static property and shapes of their input tensors. After this process, all the tensors with static shapes can be identified. These shapes will remain fixed during the entire computation. Second, after a graph is partitioned onto different servers and, on each server, before the sub-graph is executed, the data buffers of the receiver-side tensors (for those whose tensor shapes can be statically determined) or their meta-data buffers (for those whose tensor shapes cannot be statically determined) are preallocated. The remotely accessible addresses of these buffers are then passed to the servers holding the upstream tensors that they depend on. The delivered addresses are then set associated with the corresponding sender-side graph nodes responsible for transferring these tensors.

On the sender, a memory buffer to be transferred through RDMA needs to be registered beforehand to the RDMA NIC to allow its access. This registration process involves OS kernel actions such as pinning the buffer as non-pageable and therefore introduces extra overhead. In addition, the allowed number of registered buffers is bounded by the specific RDMA hardware. Therefore, simply registering the data buffer of each tensor on demand when it needs to be transferred to remote server could introduce significant overhead and might experience unexpected errors due to hardware resource limit. A more appropriate way of managing the RDMA-accessible memory is to preallocate a large enough memory buffer to register once to RDMA NIC. The size of the preallocated memory is determined by our graph analyzer. A memory allocator is used to manage the preallocated memory.

Decide tensor allocation site. Normally, a sender of a tensor through RDMA would need to allocate an extra RDMA-accessible buffer and copy the tensor from the original buffer to it. To avoid this memory copy, the graph analyzer would prefer to allocate the RDMA-accessible buffer directly for the to-be-transferred tensor. One challenge to achieve this is to find out when a specific tensor is allocated (or its allocation site): the actual storage of an input tensor of a graph node might not be allocated at the execution of its direct predecessor node, because some graph nodes may conduct in-place manipulation on their input tensors, and hence a tensor buffer may be passed through multiple nodes on a path in the graph. We therefore propose a dynamic analysis method to address this.

In order to get the allocation site of the storage of a tensor that is to be transferred, the graph analyzer instruments the tensor allocator used in the graph execution runtime. During the execution of the graph for the first mini-batch iteration, for each tensor allocation, it records the data buffer address of the tensor and the information of the corresponding graph node that invokes this allocation into a map with the tensor buffer address as the key. This node information includes the identification of the graph node and the id of the allocation of this node; e.g., the i^{th} invocation of allocation in the execution of the node. If the information with the same address already exists in the map, the new information overwrites the old one. This way, we always keep the latest information with the same tensor address. When a graph node transfers a tensor during the execution, the runtime gets the tensor buffer address and looks up the map to get the information of the graph node that allocates the tensor buffer. It then stores the information of the tensor-allocating graph node into the set S of memory regions that should ideally be allocated in the RDMA-accessible region directly. During the graph execution of the subsequent mini-batches, for each tensor allocation, the runtime checks whether the executing graph node exists in set S. If so, it allocates a tensor buffer from the allocator that manages the RDMA-accessible memory regions; otherwise, it allocates the tensor buffer from the normal allocator. This way, the data buffer of a to-be-transferred tensor, captured in S, is naturally RDMA-accessible without the need of extra copy.

3.5 GPUDirect RDMA

GPUDirect RDMA is a technology that allows an RDMA NIC to access GPU memory directly, without going through host memory, thereby offering the opportunity to save memory copy to GPU. With the design principle and methodology in our work, applying GPUDirect RDMA is straightforward because, at user-level, it similarly just needs to allocate a GPU memory space in a mapped pinned mode through the CUDA API [1] and register to the RDMA NIC, and the graph analyzer can decide which tensors need to be allocated in the same way as described in §3.4.

It is relatively tricky though to poll a value in GPU memory efficiently. Issuing a GPU kernel for every polling at an address may incur too much kernel-launch overhead, and using a kernel function to poll an address repeatedly until the state becomes ready will waste the precious GPU computing resources. We therefore always employ the mechanism with dynamic allocation described in Section 3.3 for tensor transfer through GPUDirect RDMA. Specifically, the meta-data of a tensor can be maintained in host memory so the polling only happens at the CPU side, while the actual tensor data can be stored in GPU memory and transferred through one-sided RDMA read.

4 Implementation

We implement our techniques in TensorFlow (r1.2) [3], a popular open-sourced deep learning framework in community and industry. Our implementation contains about 4,000

lines of C++ code, where the RDMA communication library (using the `libibverbs` API on Linux) takes about 1,800 lines and the rest are modifications to TensorFlow including the graph analyzer. All those changes are transparent to users.

TensorFlow organizes a deep learning computation as a data-flow graph. Users first build a graph through its high-level Python or C++ interfaces and then initiate the deep learning computation through associating the graph with a runtime session. The graph is composed of tensors and operators. The operators refer to the computing operations of the corresponding graph nodes, while the tensors represent data flowing through the edges connecting the nodes. Users are also allowed to develop customized operators and add those into the graph. Unlike the normal computational operators that are added in graph by users during the graph build phase, `Send` and `Recv` operators, which are used to transfer tensor data along edges across graph partitions, are added in the graph by the framework and are transparent to users.

To implement the mechanisms of transferring tensor data over RDMA as described in §3, we develop two pairs of custom operators and introduce an extended scheduling mechanism. For transferring tensors with a static placement, we implement the `RdmaSend` and `RdmaRecv` operators. During the graph analysis phase, the receiving tensor is preallocated with RDMA-accessibility and set as a property of `RdmaRecv`. This tensor is never freed until the entire computation finishes, so its address never changes in the entire computation. The remote-accessible address of the tensor is then passed to the server that holds the corresponding `RdmaSend` operator and set as its property. Once `RdmaSend` is scheduled to execute, it directly updates the content of the receiving tensor through a one-sided RDMA write. There is no need for some special mechanism to notify `RdmaSend` that the transferred tensor has been consumed by `RdmaRecv` because the next scheduled execution of `RdmaSend` is naturally guaranteed to happen after the consumption of the received tensor due to the control dependency of the loop in the graph or the execution sequentiality of multiple mini-batch iterations. Similarly, we also implement another pair of operators, `RdmaSendDyn` and `RdmaRecvDyn` to support tensor transfer with dynamic allocation as described in §3.3.

TensorFlow originally supports two types of execution modes for operators: synchronous and asynchronous. For both types of operators, once an operator is popped out of the ready queue to execute, it simply completes its execution synchronously or asynchronously without the need to be enqueued into the ready queue again. However, the `RdmaRecv` and `RdmaRecvDyn` need to poll the flag byte in the data or meta-data buffer of the receiving tensor. If executing totally away from the scheduling mechanism, it either suffers from busy loop wasting processor resources or long latency due to periodic sleep. We therefore introduce a new execution mode of operator called *polling-async*. The execution of this type of operator contains two phases. When executing in the polling phase, the scheduler checks whether the polling succeeds. If not, it simply re-enqueues this operator into the tail of the ready queue; otherwise, it changes the execution mode of the operator to asynchronous and reschedules the execution. This way, we reduce the polling overhead when there are other ready operators to execute.

Type	Benchmark	Model size (MB)	Variable Tensor#	Computation time (ms)
CNN	AlexNet	176.42	16	7.61 ± 0.29
	Inception-v3	92.90	196	68.32 ± 0.73
	VGGNet-16	512.32	32	30.92 ± 0.19
RNN	LSTM	35.93	14	33.33 ± 0.24
	GRU	27.92	11	30.44 ± 0.32
FCN	FCN-5	204.47	10	4.88 ± 0.28

Table 2. Deep learning benchmarks (Note: the LSTM and GRU are configured with hidden vector size of 1024 and step size of 80; the FCN-5 consists of 3 hidden layers with dimension of 4096 and two layers of input and output)

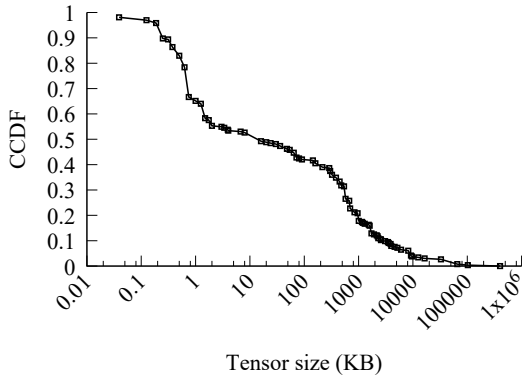


Figure 7. The complementary cumulative distribution of variable tensor sizes

5 Evaluation

We evaluate our techniques on a cluster that consists of 8 servers. Each server is equipped with dual 2.6 GHz Intel Xeon E5-2690v4 14-core CPU, 512 GB memory, 2 NVIDIA Tesla P100 GPU, and a 100 Gbps InfiniBand (IB) network adapter (Mellanox MT27700) for interconnection. All the servers are installed with Ubuntu 16.04, CUDA 8.0, and cuDNN 6. As GPUDirect is only available for some GPUs with certain system restrictions like NIC must under the same PCIe switch, most of our experiments are evaluated without GPUDirect except the last experiment in §5.2. TensorFlow (since version r1.0) supports RDMA in the way of wrapping an RDMA communication layer with the gRPC abstraction, and hence has to maintain private message buffers and incur an extra memory copy. It also relies on some IB-specific features and can only run on an IB cluster, while our RDMA mechanism

can also work with RoCE (RDMA over Converged Ethernet) network adapters. In our experiments, we empirically set 4 CQs per device and 4 QPs for each connection, through using a sufficiently large number to achieve good parallelism and communication efficiency following the guideline in [20].

Our extensive performance evaluation uses a set of representative deep learning benchmarks, including AlexNet [24], Inception-v3 [31], VGGNet-16 [29], LSTM [18], GRU [13] and FCN-5, covering convolutional neural network (CNN), recurrent neural network (RNN), and fully connected neural network (FCN). Table 2 lists some characteristics of these benchmark workloads. The model size is the sum of the sizes of all the variable tensors in a neural network, which corresponds to the communication volume between workers and parameter server processes in each mini-batch. The local computation time represents the average execution time of processing one sample data in the single-server setting. We therefore use the model size of each benchmark to characterize its network load and use the local computation time to characterize its computation complexity. These benchmarks cover both computation-intensive and network-intensive workloads. For example, the Inception-v3 model is a typical computation intensive workload, while the VGGNet-16 is mainly bottlenecked in network because each worker needs to transfer more than 1 GB (2×512.32MB) model and gradient data in each mini-batch. Among these benchmarks, the sizes of variable tensors vary from tens of bytes to hundreds of megabytes. In many cases, the existence of large tensors may substantially influence communication behavior. Figure 7 shows the distribution of number of tensors with different tensor sizes in our benchmarks. As shown in the figure, more than 50% of the variable tensors are larger than 10KB, and more than 20% are even larger than 1MB. In terms of the total capacity, the tensors that are larger than 1MB occupy 96% of the capacity among all tensors.

We conduct most of the experiments on synthetic datasets that are randomly generated and mainly used for evaluating the execution time. To demonstrate the effect of our techniques in real scenarios, we also evaluate the convergence of 3 end-to-end applications on a real-world datasets, which includes a translation task (Seq2Seq) using the sequence-to-sequence model [30] on WTM'10 French-English machine translation corpus [5] containing about 20 GB text data in total, an image recognition task (CIFAR) using the CIFAR-10 model on its public dataset [23] consisting of 60,000 32×32 colour images in 10 classes, and an RNN based sentence embedding task (SE) used in our real production. We use a private production dataset containing about 3.7 GB text data in this model.

All performance numbers with respect to throughput (mini-batches/second) in our experiments are calculated by averaging among 5 runs with each processing 100 mini-batch iterations. In all cases we observed very little variation, thus we omit the error bars in all figures.

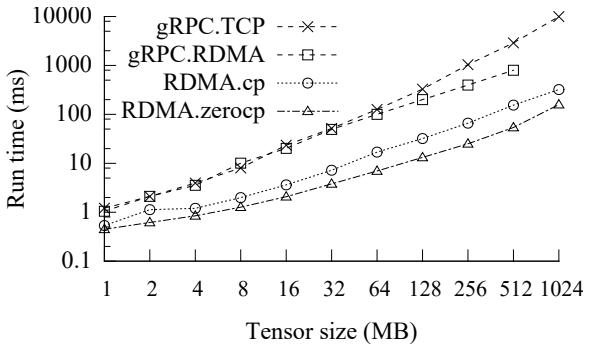


Figure 8. The performance comparison on a send/receive micro-benchmark between two servers.

5.1 Performance on Micro-benchmark

In order to understand the direct benefit of our design on system performance, we first evaluate our tensor transfer mechanism over RDMA using a micro-benchmark.

We set up two servers only to perform a tensor transfer, so as to compare our network performance with gRPC over TCP and gRPC over RDMA. The receiver also performs a lightweight `reduce_max` operator to consume the passed tensor. Figure 8 shows the efficiency of transferring tensors with different sizes. We first compare our RDMA-based mechanism (i.e., RDMA.zerocp) with the TensorFlow's original gRPC-based solutions, including both the gRPC over TCP (i.e., gRPC.TCP) and the gRPC over RDMA (i.e., gRPC.RDMA). As shown in the figure, our mechanism can outperform both of them significantly. For example, RDMA.zerocp can improve the speed by 1.7× to 61× over gRPC.TCP for different message sizes. For gRPC.RDMA, even though it adopts an RDMA protocol under gRPC, it still needs to conduct data serialization/de-serialization and data copy between RDMA pinned buffer and tensor memory on both the sender and the receiver. In contrast, our RDMA-based mechanism can completely avoid any data copy and serialization overhead, and hence achieves 1.3× to 14× performance improvement compared to gRPC.RDMA for different message sizes. Note that, there is a missing data point for gRPC.RDMA at message size of 1GB, because TensorFlow with gRPC.RDMA will crash when the transferring data size is larger than 1GB. To evaluate the memory copy overhead, we manually turn off our graph analysis optimization, so that the tensor data in the sender is unable to be pre-allocated as RDMA-accessible. To perform tensor transfer, the `RdmaSend` operator has to allocate a new RDMA-accessible buffer, copy the tensor into it, and then conduct the actual RDMA write. The curve of RDMA.cp in Figure 8 demonstrates the performance of this case. As it shows, RDMA.zerocp outperforms the RDMA.cp by 1.2× to 1.8× for different message sizes. Note that this improvement is far less than the gap between gRPC.RDMA

and RDMA.zerocp, because RDMA.cp mechanism only involves the data copy on the sender and does not incur any data serialization/de-serialization overhead.

5.2 Performance on Deep Learning Benchmarks

This section evaluates our system on real deep learning applications. Benchmarks listed in Table 2 are evaluated on synthetic data for performance, while the 3 aforementioned applications on real datasets are evaluated for convergence. By default, experiments are configured as running in distributed settings with data-parallelism, where each machine runs a worker process and a parameter server process. During execution, each worker executes a data-flow graph replica on a portion of training data. The variable tensors are shared across workers and are placed in parameter servers in a round-robin fashion. The worker runs multiple iterations until some convergence condition is satisfied or a maximum iteration number is reached. In the following discussion, unless stated explicitly, we always compare our fully-optimized RDMA mechanism with other alternative solutions.

Performance. We run the 6 deep learning benchmarks with a synthetic dataset on the same cluster. To understand better the computation and communication behavior, our synthetic datasets are generated on the fly, which can avoid the overhead of data loading from disk. In deep learning applications, the mini-batch size is a critical hyper parameter that affects both the convergence rate and the computation time. In distributed training, we can amortize the communication overhead by using large mini-batch size, because it can increase the local computation time. However, a large mini-batch size is harmful to convergence, because it reduces the model synchronization frequency across different workers [28]. In practice, the optimal setting is tuned by users, and searching for best mini-batch size is out of the scope for this paper. In our experiments, we evaluate each benchmark with mini-batch sizes ranging from 1 to 64 (128 for some). Note that, these numbers are mini-batch sizes of a single worker. The actual mini-batch size across all workers needs to further multiply the number of workers.

Figure 9 plots the performance of TensorFlow with gRPC and with our RDMA mechanism. In general, the average improvements from using RDMA against gRPC.RDMA range from 117% up to 145% for VGGNet-16. The improvements observed for other benchmarks reach 169% for AlexNet, 65% for Inception-v3, 151% for FCN-5, 118% for LSTM, and 69% for GRU. And the improvements over gRPC.TCP are much greater; for example, 25× for VGGNet-16. For these benchmarks, the graph analyzer can statically infer all the shapes of transmitted tensors, thus the transmissions are using the static-placement mechanism.

As shown in the figure, among these benchmarks, AlexNet, VGGNet-16, and FCN-5 get relatively more significant improvements from RDMA than others, because they are mainly bottlenecked at communication. Their execution time is stable under different mini-batch sizes, because the volume of transferred data (i.e., the model size) is irrelevant to the mini-batch size and the GPU's massive computing threads can complete large mini-batches within the same time as processing the small ones. However, for other benchmarks like the Inception-v3, LSTM, and GRU, when we increase the mini-batch size to larger than 32, their local computation time also increases and becomes dominant in the overall execution time. In those cases, the gaps between gRPC and our RDMA decrease as expected.

Convergence. To demonstrate the performance gain in real scenarios, we further evaluate three end-to-end training tasks, including a translation task (Seq2Seq) using the sequence-to-sequence model [30], an image recognition task (CIFAR) using the CIFAR-10 model [23] and a sentence embedding task (SE) based on two RNN models. We use perplexity value for Seq2Seq model and loss value for others to measure the convergence quality. For each task, we randomly partition their dataset into 8 workers. Each worker continuously loads the sample data from local disk in parallel with the training process. For each model, we compare gRPC.TCP, gRPC.RDMA, and our RDMA mechanism on the same training dataset until convergence.

Figure 10 plots the convergence curves for the three models with different communication mechanisms. For the Seq2Seq model in Figure 10(a), it takes about 220 minutes to converge to perplexity under 20 with gRPC.TCP. However, when using our RDMA mechanism, it takes only 66 minutes, about 3× speedup. Even comparing to gRPC.RDMA, our RDMA mechanism achieves 53% performance improvement. Similar results can be observed in the CIFAR model (Figure 10(b)) and the SE model (Figure 10(c)). For the CIFAR model, our RDMA mechanism can speed up convergence by 2.6× compared to gRPC.TCP, and 18% to gRPC.RDMA. Finally, for the SE model, we fail to collect the results of gRPC.RDMA because TensorFlow crashes when using gRPC.RDMA. If just using gRPC.TCP, the SE model can converge to loss value of 4.5 within 185 minutes. However, our RDMA mechanism takes only about 100 minutes to converge to the same point, which speeds up the training process by 85% in total.

Scalability. Scalability is one of the most important metrics for distributed training. We evaluate the scalability of TensorFlow with both our RDMA mechanism and the original solutions on all the deep learning benchmarks with synthetic datasets. We set mini-batch size to 32 for all experiments.

The scalability of each benchmark is mainly determined by its computation and communication behavior. Figure 11 shows the scalability results of three representative workloads among all the benchmarks. We can see that different

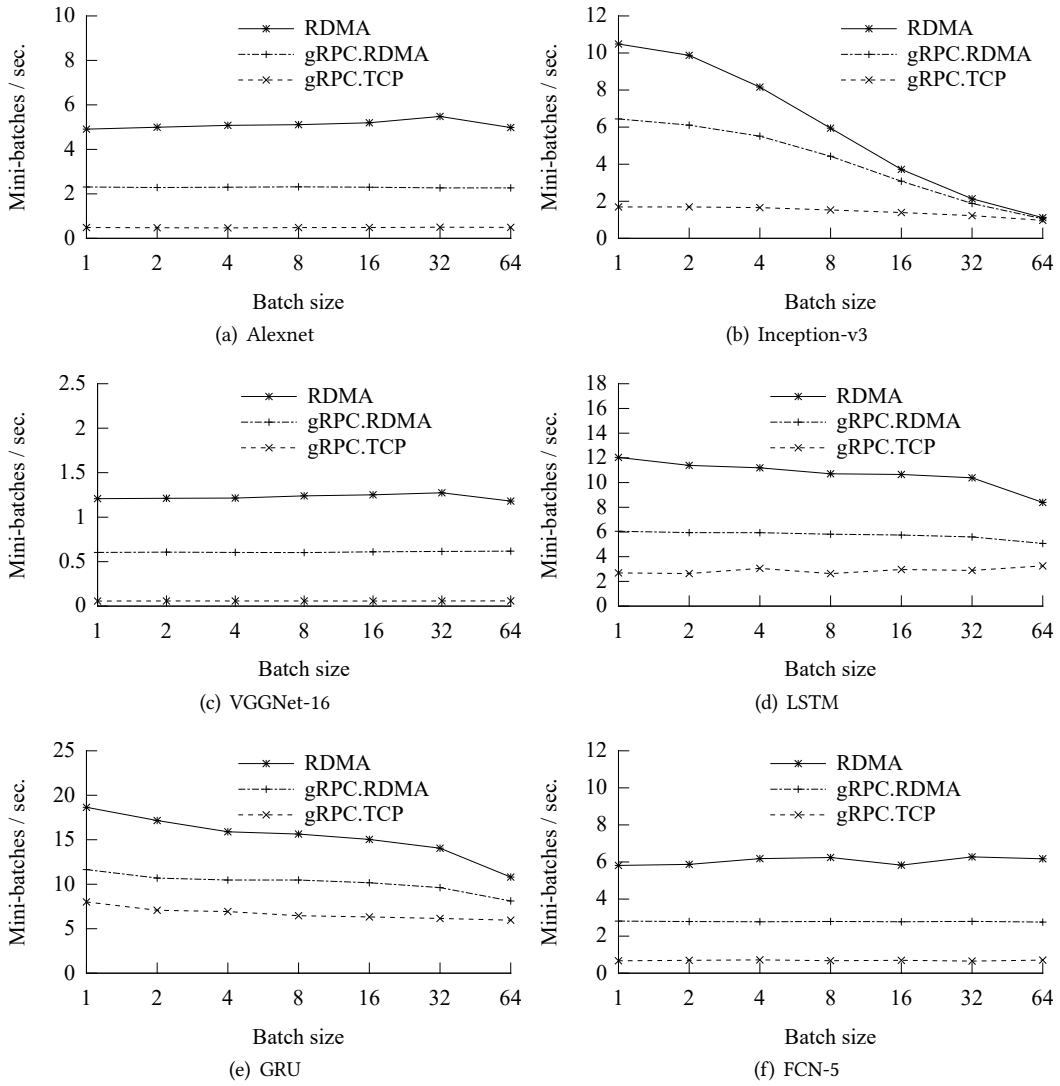
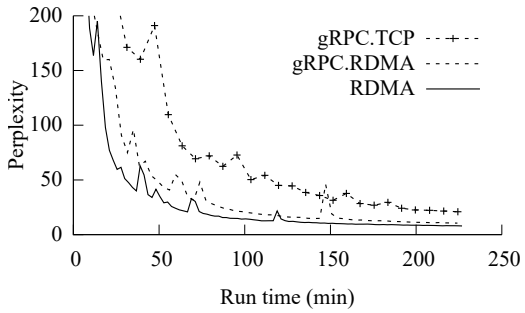


Figure 9. Comparisons with gRPC-based solutions in TensorFlow.

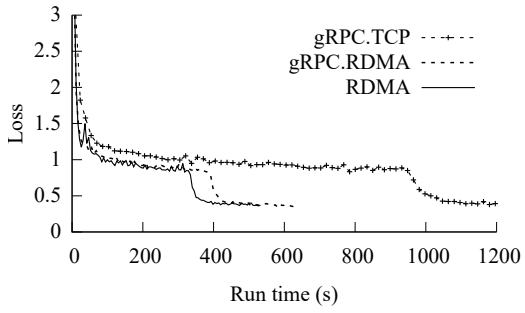
benchmarks have very different scalability patterns. For the LSTM and Inception-v3, because they are computation intensive at mini-batch size 32, we can always observe good scalability no matter whether we use our RDMA mechanism or gRPC. For example, the speedup on 8 servers for both RDMA solutions on the two benchmarks are larger than 7× against their single server cases (still involving communication between workers and parameter server processes on the same machine). Even in those cases, our RDMA remains much better than gRPC based RDMA in terms of throughput (i.e., 98% higher for LSTM and 12% for Inception-v3). For VGGNet-16, because it is a communication intensive application, its scalability is highly determined by the underlying network efficacy. In this case, our RDMA can still get 5.2× speedup against its single server, consistently remains more than 140% faster than gRPC based RDMA on different scales.

For each of these benchmarks, we also measure the throughput of its pure local implementation (the "Local" line in Figure 11), which does not involve any communication overhead. As shown in the figure, for the gRPC.RDMA case, the speedups on 8 servers relative to the local implementations for LSTM and Inception-v3 are 1.5× and 6×, respectively. It needs 4 servers to outperform the local implementation for LSTM and 8 servers for VGGNet-16 due to its much more serious communication bottleneck. In contrast, with our RDMA, all the three distributed benchmarks can outperform the local implementations with only 2 servers. The speedups on 8 servers are 5×, 7.9×, and 4.3× for the three benchmarks, respectively.

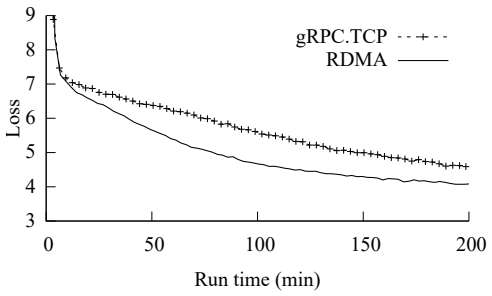
Memory Copy Overhead. We also evaluate the performance gain of removing the sender side memory copy, as described



(a) Seq2Seq

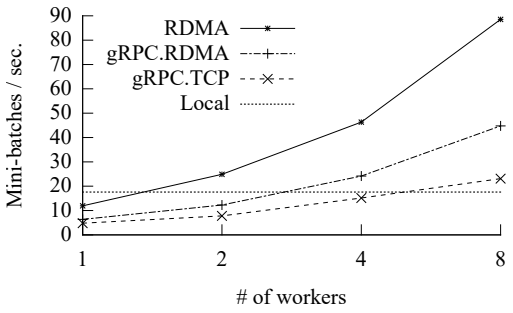


(b) CIFAR

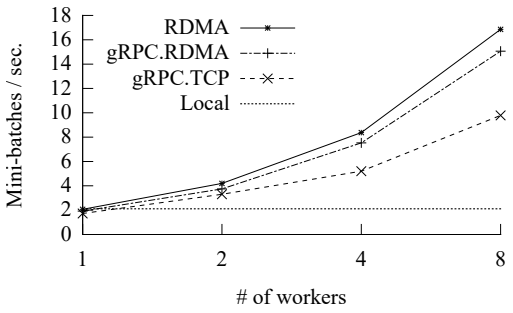


(c) SE

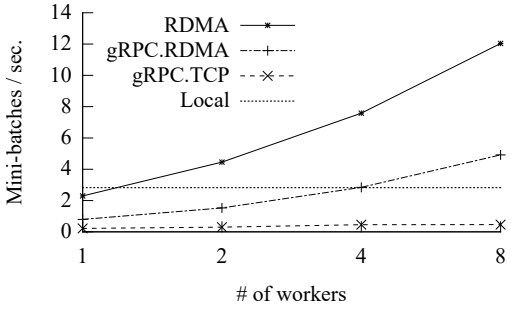
Figure 10. Convergence of real applications on TensorFlow with different communication mechanisms.



(a) LSTM



(b) Inception-v3



(c) VGGNet-16

Figure 11. Scalability of TensorFlow with gRPC-based solutions vs. RDMA.

in Section 5.1, in deep learning benchmarks. We manually turn off the graph analysis phase so as to skip the optimization for zero copy, and compare its performance with the optimized one. Figure 12 shows the average mini-batch time of each benchmark with (or without) memory copy. In general, for different workloads, the zero-copy optimization can bring up to 21% performance improvement with mini-batch size of 8. However, for some benchmarks such as the Inception-v3 and GRU, the performance gain is relatively small. This is mainly due to two factors. First, as we explained before, these benchmarks are mostly computation intensive, which could benefit little from network related optimization. Second, from the result of micro-benchmark in Section 5.1, the gain of zero copy is more significant for larger tensor size, however, the Inception-v3 includes many small tensors: the model contains 196 variables, but the total model size is only 92.9MB.

GPUDirect Support. Finally, we evaluate the performance with GPUDirect RDMA enabled for different applications, as shown in Table 3. After enabling GPUDirect, our RDMA further improves the performance by up to 54%. Improvements vary from application to application, similar to what we observed in previous experiments.

6 Related Work

Systems leveraging RDMA. With the advent of the emerging RDMA technology, a large body of research has been done on improving the performance of various distributed systems to leverage its low latency and high bandwidth. A

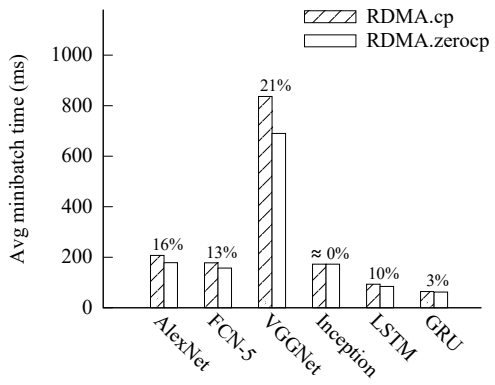


Figure 12. The memory copy overhead in deep learning benchmarks.

Benchmark	RDMA	RDMA+GDR	Improv.
AlexNet	178.5	135.2	32%
FCN-5	157.0	101.9	54%
VGGNet	690.1	610.4	13%
Inception	172.5	171.9	0.4%
LSTM	84.4	68.1	24%
GRU	62.3	52.6	19%

Table 3. The average minibatch time (ms.) and improvements with GPUDirect RDMA(GDR) in deep learning benchmarks. (8 workers)

series of efforts [16, 19, 26, 32, 35] target to optimize key-value storage systems, while some others focus on improving the throughput of distributed transaction processing systems [10, 17, 35]. FaRM [16] uses one-sided RDMA reads for key-value lookups while employing a messaging primitive for updates. This messaging mechanism uses a fixed ring buffer on the receiver side to hold received messages, and hence may bring extra overhead of copying messages to the application buffers. In addition, large messages may have to be split on the sender side and re-assembled on the receiver side due to the limited size of the ring buffer, which introduces more copying overhead. HERD [19] embraces an RPC abstraction for key-value lookup to avoid multiple remote accesses on a hash-table structure. Kalia *et al.* [20, 21] further improves it by considering lower-level factors in the RDMA hardware and optimizing it for the all-to-all cases used in transaction processing. All these research efforts target the scenarios dominated by small messages, where latency is the major objective of optimization.

Grappa [27] and GraM [37] explore the use of RDMA to accelerate distributed graph computation. In graph processing, it is appropriate to use RPC to batch many small random remote accesses caused by the complex and sparse graph structure. However, in the deep learning scenarios, a common pattern is to access dense, often relatively large, tensors.

GPUNet [22] proposes a socket-like abstraction over RDMA, which allows GPU kernel functions to communicate directly through network. Their design targets scenarios of general distributed computations on GPUs. It is interesting to look at how to integrate this level of "directness" into a dataflow-based deep learning framework with techniques in our work.

Distributed machine learning systems. Many systems have been designed to support efficient distributed computation of traditional machine learning algorithms, which usually employ shallow model structure and do not necessarily express their computation as data-flow graph, such as Petuum [39] and Parameter Server [25]. These shallow model structures often lead to sparse matrix computations, which share the similar patterns to graph processing [38]. These systems scale out by employing a parameter server architecture, which uses a set of servers to manage shared state that is updated by a set of parallel workers. This parameter server architecture can also be used to support some distributed deep learning frameworks, such as DistBelief [15], Project Adam [12], MxNet [9], and so on. Although these frameworks, unlike TensorFlow, only use data-flow graph to represent local computation at each worker, the principle and methodology in our work can also be applied. For example, as long as the data transmitted in these frameworks has a dense structure, it is straightforward to leverage our techniques to improve communication efficiency and scalability further.

7 Conclusion

The emerging deep learning workloads and network technologies such as RDMA have prompted us to rethink the widely used RPC abstraction for network communication. We observe that the abstraction does not allow application-level information to be passed to the network layer for optimizations, leading to unnecessary additional memory copy and significant performance penalty. By designing a simple "device"-like interface, along with a combination of static analysis and dynamic tracing , we have enabled cross-stack optimizations for general deep neural network training to take full advantage of the underlying RDMA capabilities, leading to up to almost an order of magnitude speedup for representative deep learning benchmarks over the default RPC library and up to 169% improvement even over an RPC implementation optimized for RDMA.

Acknowledgments

We thank our shepherd Sonia Ben Mokhtar and the anonymous reviewers for the valuable comments and suggestions.

References

[1] Retrieved in 2017. CUDA Driver API. http://docs.nvidia.com/cuda/cuda-driver-api.

[2] Retrieved in 2017. gRPC - An RPC library and framework. https://github.com/grpc/grpc.

[3] Retrieved in 2017. TensorFlow. https://github.com/tensorflow/tensorflow/tree/r1.2.

[4] Retrieved in 2017. The Apache Thrift. http://thrift.apache.org.

[5] Retrieved in 2017. WTM'15 Machine Translation Dataset. http://www.statmt.org/wmt15.

[6] Retrieved in 2017. ZeroMQ. http://zeromq.org/.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[8] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* 2, 1 (Feb. 1984), 39–59. https://doi.org/10.1145/2080.357392

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2016. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *NIPS Workshop on Machine Learning Systems (LearningSys)*.

[10] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/2901318.2901349

[11] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. *arXiv:1606.07792* (2016). http://arxiv.org/abs/1606.07792

[12] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[13] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). http://arxiv.org/abs/1406.1078

[14] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. USENIX. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[17] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[22] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 201–216. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim

[23] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[25] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[26] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX, 12. http://dl.acm.org/citation.cfm?id=2535461.2535475

[27] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[28] Jorge Nocedal Mikhail Smelyanskiy Nitish Shirish Keskar, Dheevatsa Mudigere and Ping Tak Peter Tang. [n. d.]. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *5th International Conference on Learning Representations (ICLR 17)*.

[29] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[30] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112. http:

//dl.acm.org/citation.cfm?id=2969033.2969173

[31] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). http://arxiv.org/abs/1512.00567

[32] Tyler Szepesi, Benjamin Cassell, Bernard Wong, Tim Brecht, and Xiaoyi Liu. 2015. Nessie: A Decoupled, Client-Driven, Key-Value Store using RDMA. *Technical Report CS-2015-09, University of Waterloo, David R. Cheriton School of Computer Science.* (2015).

[33] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[34] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[35] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[36] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[37] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*. ACM, 14. https://doi.org/10.1145/2806777.2806849

[38] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. TuX2: Distributed Graph Computation for Machine Learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 669–682. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/xiao

[39] 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. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15)*. ACM, 10. https://doi.org/10.1145/2783258.2783323

[40] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report. https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/