Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems

Nikoli Dryden*†, Naoya Maruyama*, Tim Moon*, Tom Benson*, Andy Yoo*, Marc Snir†, Brian Van Essen*
*Lawrence Livermore National Laboratory
{maruyama3,moon13,benson31,yoo2,vanessen1}@llnl.gov
†Department of Computer Science
University of Illinois at Urbana-Champaign
{dryden2,snir}@illinois.edu

***Abstract*—We identify communication as a major bottleneck for training deep neural networks on large-scale GPU clusters, taking over 10x as long as computation. To reduce this overhead, we discuss techniques to overlap communication and computation as much as possible. This leads to much of the communication being latency-bound instead of bandwidth-bound, and we find that using a combination of latency- and bandwidth-optimized allreduce algorithms significantly reduces communication costs. We also discuss a semantic mismatch between MPI and CUDA that increases overheads and limits asynchrony, and propose a solution that enables communication to be aware of CUDA streams. We implement these optimizations in the open-source Aluminum communication library, enabling optimized, asynchronous, GPU-aware communication. Aluminum demonstrates improved performance in benchmarks and end-to-end training of deep networks, for both strong and weak scaling.**

***Index Terms*—Deep learning, machine learning, communication optimization, collective algorithms, HPC**

I. Introduction

With the success of deep learning, accelerating the training process has become increasingly important, particularly as model complexity and dataset sizes grow. Many training toolkits have emerged that leverage multiple GPUs, either on a single node or across multiple nodes [1]–[7]. Simultaneously, large clusters of GPUs have begun to be deployed and leveraged for training deep networks. Efficiently utilizing such systems requires careful optimization of many aspects of the training process. In particular, reducing communication overheads stands out as one of the most significant requirements, especially in order to scale to large node counts.

Many of the current approaches to distributed training can be broadly divided into model- and data-parallel techniques. In model-parallel techniques, a neural network layer is partitioned across multiple processors. This is typically applied to fully-connected layers [1], where it is essentially a distributed matrix product, but it has also been demonstrated for locally-connected layers [8]. Scaling matrix products is a well-studied problem in numerical linear algebra [9], [10]. In data-parallel techniques, layers are replicated and a mini-batch's data is partitioned across multiple processors, which perform forward and backward propagation independently before synchronizing their parameter updates. This is the typical approach to distributed training for convolutional layers, and is also often applied to entire networks.

Scaling data-parallelism typically relies on increasing the size of the training mini-batch, as scalability is ultimately limited by the number of samples in each mini-batch. Additionally, larger mini-batches help ensure that each processor is efficiently utilized. Increasing mini-batch size is non-trivial, as it can impact the quality of the learned model and has a complex interplay with the model's learning rate [11]–[14]. Several techniques have demonstrated successful large mini-batch training, including linear warmups [15] and layer-wise adaptive learning rates [16], typically for image classification problems. It remains to be seen how general these approaches are, especially when applied to non-image data.

The communication requirements for data-parallel training are particularly large due to the need to synchronize parameter updates. This operation is a global allreduce operation on each layer's parameters. As modern networks often have large numbers of parameters and many layers, this allreduce is a significant cost, and communication has been consistently identified as a major bottleneck in scaling [17], [18]. The allreduce is typically implemented either via centralized parameter servers, which accumulate and distribute updates from workers; or in a distributed manner via an operation like `MPI_Allreduce`. Some systems make use of sparse, quantized, or compressed communication to reduce communication overhead [19]–[21]; we view these approaches as complementary to our work.

In this work, we study the communication requirements for training modern deep networks and identify implementation techniques to reduce them. Our focus is on distributed GPU systems using CUDA and MPI, where all nodes are interconnected with a high-speed network. This is typical of modern GPU supercomputers.

We begin by examining communication overheads for both strong and weak scaling of training. Using ResNet-50 [22] as our example, we find that even at small scales, communication accounts for a significant portion of total runtime and the

overhead worsens rapidly as training is scaled onto more GPUs. This is exacerbated for strong scaling, where the volume of work per processor decreases with scale while the cost of communication increases, making strong scaling to large numbers of GPUs unprofitable. Weak scaling fares better, but communication overheads still prevent optimal scaling, and it yields poor improvements on many GPUs. We then turn to alleviating the communication overheads.

Overlapping communication and computation is a standard approach to help hide communication overheads. The standard formulation of backpropagation and gradient descent for training deep networks enables communication to be overlapped with no algorithmic changes, and we show that when done well, this can significantly reduce communication overheads. To maximize overlap, we aim to begin communication as soon as possible: whenever a layer has finished computing its updates, an allreduce for it is started. This leads to relatively fine-grained communication and requires quality implementations of non-blocking communication. Since communication is done for each individual layer, the volume of data being communicated in each operation is quite small. This results in many of the allreduces being latency-bound rather than bandwidth-bound, contrary to the typical case for training deep networks. Latency also becomes increasingly important at at large scales.

Once latency becomes a significant factor in communication performance, local synchronization overhead also becomes a concern. The standard approach to interfacing CUDA-aware MPI for communication with data being computed on GPUs is to synchronize the stream computing the data prior to beginning communication. This imposes overheads both due to synchronization and because kernel launch latencies for GPU computations cannot be pipelined as effectively. Further, performing this synchronization blocks the host, limiting host/GPU overlap, which is especially important for hiding I/O costs. We propose instead to make our communication operations aware of the stream a GPU buffer is being computed on. This enables them to function similarly to a CUDA kernel, and minimizes the synchronization overheads without impacting pipelining or overlap. Unfortunately, current MPI distributions, even those that are CUDA-aware, do not provide a means to do this.

These improvements enable the communication overhead for training deep networks to be significantly reduced and training to be scaled to larger systems. We summarize our contributions as follows:

- We examine the communication overheads involved in training deep networks and show that overlapping can significantly reduce them.
- We identify the importance of getting good performance for fine-grained, often latency-dominated communication. We show how latency-optimized allreduce algorithms can significantly outperform the more common bandwidth-optimized ring algorithms for relevant data sizes, especially at scale.
- We demonstrate techniques to perform communication on GPU data in a non-blocking manner for both the host and GPU, while reducing synchronization overheads.
- We introduce the Aluminum library, an open-source library[1] that implements our communication techniques and provides a generic interface to communication substrates. Its API is similar to MPI's and it can be used as a replacement for existing libraries with trivial changes.
- We evaluate the impact of these methods in both microbenchmarks and end-to-end training within the open-source LBANN toolkit [1].

II. Communication requirements

We begin by discussing in more detail the communication involved in training a deep network, including where the communication occurs and what volume of data is moved. This forms the basis of our subsequent discussion on optimizing communication.

A. Where and what is the communication?

Training a deep network can be thought of as involving three phases that are repeated iteratively: forward propagation, backpropagation, and optimization. Forward propagation involves computing the output of the network for the input data (essentially, inference). Backprop computes gradients to update the network parameters based on its inference, and the optimization phase applies the updates, typically using a variant of stochastic gradient descent. When using a data-parallel approach to parallelize training, communication is performed only during backpropagation[2] (see II-D for the model-parallel case). This communication is an allreduce that synchronizes the independent updates that each processor computes into a global update that can be applied independently. (See [23]–[25] for overviews of deep learning and its optimization and parallelization.)

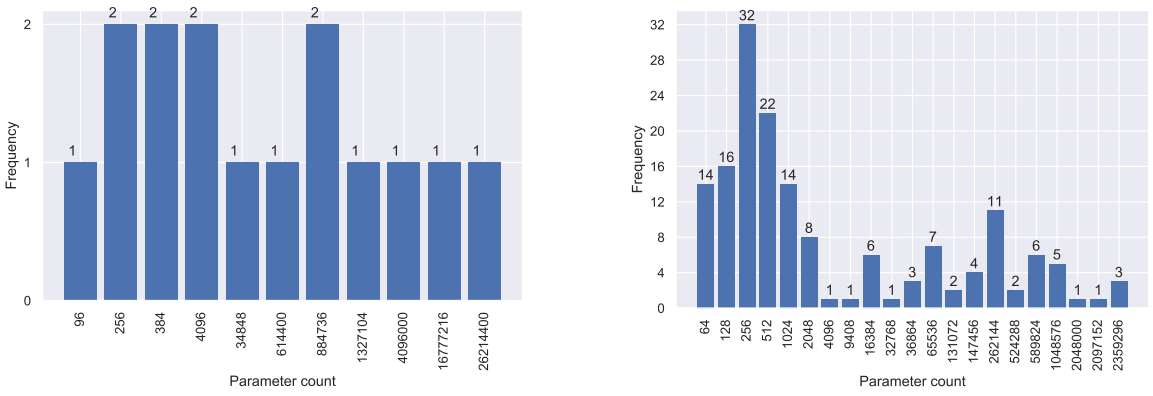
Implementations can perform this allreduce either using centralized parameter servers (e.g. as in TensorFlow) or via a decentralized allreduce implementation such as MPI's `MPI_Allreduce` or equivalent. We focus on the latter case exclusively in this work.

Backprop is performed sequentially for each layer in a network, beginning with the final layer and ending with the input layer. Each layer receives as input an "error signal" from the subsequent layer, and computes a modified error signal as its output. If a layer additionally has parameters to learn, the layer will compute a gradient based on the input error signal. It is important to note that within a layer, these two operations are independent and can be performed in any order. Once the gradient has been computed, it can be combined with other processors' gradients to compute the global gradient for that layer.

The granularity of communication can vary depending on the implementation. At one extreme, all data could be

[1] https://github.com/LLNL/Aluminum

[2] One could equivalently think of communication as being performed during the optimization phase; we choose backprop for convenience.



(a) AlexNet

(b) ResNet-50

Fig. 1. Histograms breaking down the number of parameter buffers (essentially, a layer) of a given size for the AlexNet and ResNet-50 networks. In our implementation, each parameter is a 4-byte float.

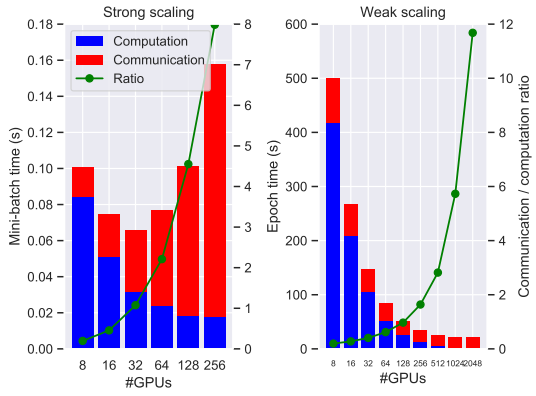


Fig. 2. Strong and weak scaling results for ResNet-50 using our synthetic benchmark on Sierra, using NCCL with no communication/computation overlap.

combined into a single buffer and allreduced once backprop completes for every layer. Alternatively, allreduces can be done as soon as the gradient computation for a layer completes, and work on a per-buffer basis. Many implementations (including ours) keep separate, non-contiguous buffers for the parameters for each layer for simplicity, so operating on a per-buffer basis is typical.

In this work, our layers use 4 byte single-precision floats to store parameters, and we communicate parameters in this format. Within the networks we consider, convolutional, fully-connected, and batch normalization [26] layers have parameters that must be learned. In our implementation, convolutional and fully-connected layers have their parameters stored in a single buffer per layer. Batch normalization, for convenience, has two buffers, one each for its scale and bias.

B. Communication volume

We now look to understand the amount of data and number of buffers that must be communicated in an iteration. This depends on the architecture of the network being trained (e.g. number and size of filters in a convolutional layer). Figure 1 plots histograms of parameter buffer size for two representative image classification networks, AlexNet [27] and ResNet-50 [22].

AlexNet is a fairly shallow network that has several large fully-connected layers, and is a commonly used baseline or building block where state-of-the-art accuracy is not necessary. It has relatively few buffers: five convolutional layers and three fully-connected layers, with all but the final layer having a separate bias. The three largest buffers are the fully-connected layers, which contain a majority of the parameters.

ResNet-50 is more representative of modern CNNs, which have many more layers, batch normalization, and fewer fully-connected layers. ResNet architectures do not have biases, but many of the small buffers are due to the parameters for batch normalization layers. Since many recent architectures and benchmarks have focused on ResNet-like architectures or ResNet-50 in particular (e.g. [28]), we will use it for the remainder of the paper.

A key observation to make from these plots is that both networks require allreduces to be performed on many small buffers. For ResNet-50, a majority of the buffers are 8 KiB or less. However, there is also a very large range of buffer sizes, spanning 256 bytes to megabytes. A single algorithm for performing the allreduce is unlikely to perform optimally for all of these sizes, as they span both latency- and bandwidth-dominated regimes. (We demonstrate this in Section VI.)

C. Communication overhead

We now empirically examine the communication overhead involved in training ResNet-50 on ImageNet [29] in various configurations. Our goal in this section is to understand the baseline performance, which can then be improved upon. We utilize a simple synthetic benchmark that incorporates the compute cost of convolutional layers (the primary computational cost in ResNet-50) and the communication cost of synchronizing layer gradients.

The compute time is determined by benchmarking the runtime of the relevant cuDNN [30] routines for convolution on the local problem size of each convolutional layer. Communication time is determined by benchmarking allreduces of the relevant sizes, using the NCCL collective communication library [31]. We assume that a separate allreduce is performed on each buffer. We treat the fully-connected layer as being model-parallel (see II-D) and neglect it for simplicity; as it is a small layer, this does not significantly affect our results. Note that this benchmark is meant to illustrate the major sources of communication and computation, and neglects many aspects of a full training pipeline, such as I/O, optimization, activation layer computation, and internal synchronization.

We run this benchmark on the Sierra supercomputer [32], which consists of 4,320 compute nodes with two IBM POWER9 CPUs and four NVIDIA V100 (Volta) GPUs with NVLINK2 per node, interconnected via a dual-rail InfiniBand EDR network. We use CUDA 9.2.148, cuDNN 7.2.1, and NCCL 2.3.

1) Strong scaling: To strong scale ResNet-50 training, we keep all parameters constant and increase the number of GPUs being trained on. The mini-batch size is 256, per the original paper. Due to memory constraints, we cannot train ResNet-50 on fewer than 8 GPUs, and the mini-batch limits us to at most 256 GPUs. We additionally neglect issues that may be caused by batch normalization having few samples per node [33], [34].

We plot the mini-batch iteration time, as well as a breakdown of computation versus communication, in Figure 2 (left). As the number of GPUs increases, the computation time decreases, but the scaling is unfortunately sublinear. Simultaneously, communication requirements increase as more nodes are involved while the number of iterations remains constant. Runtimes improve up to 32 GPUs, after which communication overheads outweigh the benefits. The communication/computation ratio rapidly increases, and even at only 32 GPUs accounts for more than half the runtime.

2) Weak scaling: For weak scaling, we keep every parameter but the mini-batch size fixed and train with 32 samples per GPU. This is the same regime as [15] or [16], which demonstrate how to maintain model accuracy despite the large mini-batch, and offers a good compromise between GPU utilization and memory requirements. Note that as the mini-batch size increases, the number of iterations to complete an epoch decreases (it is 4955 iterations when the mini-batch size is 256).

We plot total epoch time, again with a communication/computation breakdown, in Figure 2 (right). In this case, computation scales linearly. The total time for communication decreases as the number of GPUs increases, because fewer iterations are performed, resuling in fewer rounds of communication, although this trend breaks down for large numbers of GPUs. However, the ratio of communication to computation steadily worsens, resulting in a nearly 6x ratio of communication to computation on 1024 GPUs and 12x on 2048 GPUs. Despite this, it remains profitable to weak scale ResNet-50 training to this scale, though it suffers from significant diminishing returns.

D. Model-parallel fully-connected layers

We briefly discuss the differences in communication when using model-parallel fully-connected layers. These essentially implement a distributed matrix product, which can be thought of as a collective operation involving every processor. Communication is now required in both forward and backward propagation to compute the layer's output, error signal, and gradients; however, no additional communication is needed to synchronize the gradient update. Since matrix products typically require their input data to have a particular distribution (e.g. blocked), data may need to be moved from a "data-parallel" distribution for this. The communication operations performed depend on the algorithm being used, but typically involve a variety of collectives beyond allreduce.

III. Optimizations

We now discuss two basic optimizations for reducing communication overhead and improving performance: overlapping and latency-efficient allreduce algorithms. Neither of these techniques are new. Overlapping communication and computation during training has been discussed before (e.g. [15]), and we will provide additional detail on implementing them with GPUs. Latency-efficient allreduces are similarly not new [35]; however, deep learning applications have typically preferred bandwidth-optimized ring-based allreduce implementations as in the Baidu allreduce [5] or NCCL/NCCL2 [31] libraries.

A. Overlapping

Overlapping communication and computation when training deep nets involves performing gradient update allreduces concurrently with backpropagation and optimization. This can be done within the constraints discussed in Section II-A. Thus, to maximize the potential for overlapping, each layer should compute its local gradient update first and then start an asynchronous allreduce on that buffer. The remainder of backprop can be performed in the same manner, and the allreduce completed when the optimization phase for that layer begins. This enables the allreduce to be hidden by the error signal computation in the associated layer, and all computation in all remaining layers.

Achieving communication/computation overlap when running on GPUs requires additional work, as we do not want to block the CUDA stream training computations are performed on. We can instead make use of separate, internal streams to perform the communication and handle synchronization as needed.

B. Latency

While performing allreduces as soon as possible helps maximize overlap, it results in many small allreduces being performed, some as small as 64 parameters (256 bytes). This size regime is latency-dominated instead of being bandwidth-dominated, and the size of allreduces that are latency-dominated increases as the number of GPUs increases.

Typically, allreduce libraries for deep learning have been bandwidth-optimized and employ ring-based algorithms [5], [31]. These algorithms perform very well in multi-GPU shared-memory systems (especially ones optimized to have ring topologies, such as the NVIDIA DGX1) or at small distributed-memory scales despite not being latency-optimized. AlexNet-style networks (see Figure 1a) also have far fewer small allreduces and several very large allreduces.

Tree-based allreduce algorithms can offer much better performance in latency-dominated regimes [35]. Recursive-doubling is preferred for small messages, and has optimal latency. Recursive-halving/recursive-doubling (also called Rabenseifner's algorithm) has slightly worse latency, but better bandwidth utilization, and is preferred for larger messages.

To make this more precise, if α is the network latency and β its inverse bandwidth, p the number of processors, and n the buffer size, the communication time for a ring allreduce is $2(p-1)\alpha + 2\frac{p-1}{p}n\beta$. This explains the increasing communication time in Figure 2. The bandwidth term remains nearly constant as p increases, but the latency term rapidly becomes important, especially with many small messages. In contrast, recursive-doubling has communication time $\log p(\alpha + n\beta)$, and Rabenseifner's algorithm has communication time $2\log p\alpha + 2\frac{p-1}{p}n\beta$. While Rabenseifner's algorithm has the same bandwidth term and better latency than the ring algorithm, the nearest-neighbor communication in rings often enables them to outperform it in practice for large messages.

In this work we present the Aluminum library, which augments NCCL with tree-based algorithms and dynamically select the fastest algorithm based on the buffer size and the number of processors. An additional optimization is to run multiple allreduces concurrently. In a latency-dominated regime, we are not limited by packet injection rates or similar issues, but instead by waiting for communication to complete. This enables pipelining the allreduces to further reduce communication overhead.

IV. Interfacing with MPI

Modern MPI distributions provide large suites of optimized communication algorithms, including tree-based allreduce algorithms. Many of them are also "CUDA-aware", in that they accept pointers to GPU buffers and can perform communication on them. Why can we not simply use CUDA-aware MPI directly for allreduces when appropriate? Fundamentally, we argue that because MPI is unaware of users' CUDA streams, a semantic mismatch between the MPI and CUDA programming models arises, leading to communication and computation overheads due to unnecessary synchronization. We will then discuss approaches to fixing this mismatch.

A. Problems

When using CUDA to compute data on a GPU, one typically launches a sequence of compute kernels on a CUDA stream. The CUDA runtime ensures that kernels launched on a stream are executed in launch order (there is no ordering between multiple streams unless one is imposed using explicit synchronization). This means that, provided kernels are launched in the right order, all its inputs are ready when it begins execution. Kernel launches (along with most other CUDA operations) are asynchronous and do not block the host, but there is a cost (roughly 10 µs) associated with launching them. For this reason, one typically launches many kernels in a row without waiting for their completion, pipelining the launches and hiding the launch latency for every kernel beyond the first.

MPI runtimes are unaware of users' CUDA streams. Therefore, when a user passes a GPU buffer to an MPI routine, MPI has no way to determine whether there is a pending computation on a stream that will write to the buffer. To ensure correctness when a kernel may write to the buffer, the user must synchronize the stream to complete pending computation. This forces the application into a bulk-synchronous model of separated computation and communication phases, preventing pipelining of kernel launches and overlapping of communication and computation. Similarly, when MPI communication is in progress, there is no way for a stream to wait for a blocking operation's completion (e.g. `MPI_Allreduce` or `MPI_Wait`). This further means that other streams that might synchronize with the first stream also need to be blocked.

Alternating computation and communication phases in this manner leads to an awkward and error-prone programming model, and underutilization of both the network (during computation phases) and GPU (during communication). Frequent blocking on the host also limits the ability to overlap communication and computation with other operations, such as I/O. In the context of training deep nets, I/O can be quite expensive, so hiding it is crucial. Finally, when latency-optimized communication is necessary for scaling, minimizing additional synchronization is important.

A further concern with using CUDA-aware MPI is practical. We have observed that CUDA-aware MPI runtimes often do not handle operations with GPU buffers correctly when they are performed from multiple threads, even when `MPI_THREAD_MULTIPLE` is enabled. We hope that this can be resolved by improved documentation and bug fixes by MPI distributions.

B. Possible solutions

One solution that achieves correctness is to push the synchronization into the MPI library. Since it is unaware of which user stream is producing the buffer to be communicated, the library must synchronize the entire device, either explicitly or via CUDA's default stream semantics. This resolves none of the performance issues noted above.

A more promising solution is to treat MPI communication operations as "just another kernel" to be enqueued on a stream. As a proof-of-concept, NCCL operations take a stream as an argument and employ the usual kernel launch semantics: it doesn't block the host, is ordered within the stream, and blocks the stream.

Unfortunately, MPI operations cannot take a stream parameter. However, we find it sufficient to associate a single

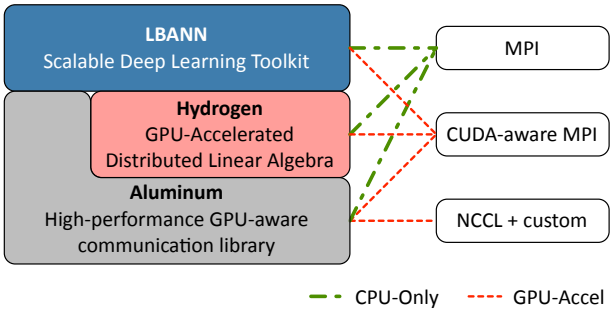


Fig. 3. Integration of Aluminum into the open-source toolkits LBANN and Hydrogen.

stream with a communicator. Every operation that uses the communicator and a GPU buffer can then assume that the buffer is written to by some kernel on that stream, and perform the appropriate synchronization with respect to only that stream. Within MPI, this association can be implemented as an attribute attached to the communicator. To achieve good performance, the implementation can then make use of fine-grained CUDA events and other synchronization, driven by a background thread, to progress communication without blocking execution. We have taken this approach and implemented it in our Aluminum library, detailed in the next section.

While this paper has focused on allreduces, due to their importance in training deep networks, these approaches are in no way exclusive to allreduces and are applicable to any communication operation.

V. The Aluminum library

We have developed the Aluminum library as an open-source communication library. It provides a generic API for communication operations implemented by multiple backends, and currently supports MPI, NCCL, and custom implementations of various operations for both CPU and GPU communication. Aluminum does not replace other communication libraries, but provides a portable layer to optimized communication substrates as well as benefiting from the ubiquity of MPI as a baseline. For example, it would be easy to support AMD's software stack with little change in applications.

This library encapsulates the proposed optimizations discussed in Sections III and IV-B, including easy non-blocking operations on both host and GPU, latency-optimized algorithms, and CUDA-friendly synchronization semantics. It is currently being leveraged by both the LBANN deep learning toolkit [1] and the Hydrogen distributed linear algebra library [36] (a fork of the Elemental library [37]) as shown in Figure 3. Section VI presents benchmarks demonstrating the effectiveness of these optimizations.

A. API and semantics

Aluminum is a C++11 library with an API inspired by MPI's. This similarity means that integrating Aluminum into existing applications should be quite simple. In particular, since NCCL and/or MPI are frequently used by distributed deep learning frameworks, they can easily take advantage of Aluminum.

TABLE I
Aluminum allreduce capabilities based on backend.

Backend	Algorithm Support	Features
MPI	Ring, recursive-doubling, Rabenseifner	Ubiquitous, optimized
NCCL	Ring	GDR, optimized for GPUs
MPI-CUDA	Ring, recursive-doubling, Rabenseifner	Host-transfer algorithm

It consists of a core providing internal implementation frameworks and three communication backends (and is extendable to support more):

MPI provides both an interface to MPI (by directly calling MPI routines) and custom collective implementations built atop of MPI. It is meant to be used with host buffers.

NCCL provides a direct interface to NCCL for use with GPU buffers.

MPI-CUDA implements a variety of custom algorithms that are built on top of MPI and CUDA for use with GPU buffers. This backend implements our "host-transfer" allreduce. (This is independent of CUDA-aware MPI.)

The backends and notable features are summarized in Table I.

The API to invoke a non-blocking, in-place allreduce (for example) looks like: `Al::NonblockingAllreduce<Backend>(buffer, count, op, comm, req)`, where `buffer` and `count` define the buffer to be reduced, `op` is a reduction operation (e.g. summation), `comm` is an Aluminum communicator object, and `req` is a request object. C++ templates are used to infer the type of the buffer and dispatch the operation to the correct backend. Aluminum also handles algorithm selection where appropriate, making a reasonable choice based on the buffer and communicator sizes (this can also be manually specified by the user). The allreduce then proceeds asynchronously, and can be completed via a wait operation: `Al::Wait<Backend>(req)`. Every backend automatically handles Aluminum's synchronization semantics, described below.

Aluminum currently supports a subset of the standard MPI collective operations in both blocking and non-blocking versions, including: reduce, allreduce, reduce-scatter, allgather, and broadcast. The MPI-CUDA backend additionally supports the basic send, recv, and sendrecv point-to-point operations for GPU buffers. The NCCL backend is currently limited to only the subset of reduction operations that NCCL supports (summation, multiplication, min, and max); our other backends support a more general set of reduction operations.

The semantics of Aluminum's blocking and non-blocking operations differs from MPI, and it implements the approach discussed in Section IV-B in a manner that provides a fairly generic interface for both CPU and GPU operations. We associate a "stream of computation" with each communicator.

For GPU backends, this is a CUDA stream. For the MPI backend, this stream is implicit, and can be thought of as the calling thread or process; this could be made explicit in the future to better support threading or lightweight threading libraries. All operations then synchronize the communicator's stream as necessary. This is critically important for GPU operations, where it means that no GPU operation blocks the host. From the example above, if the `Al::Wait` operation were used with the MPI-CUDA backend, it would (perhaps counterintuitively) not block the host, but instead block `comm`'s CUDA stream until the allreduce completed.

B. Implementation details

We now detail some of the notable implementation details for Aluminum.

1) Communication engine: Any communication that must perform operations on the host without blocking the main thread of execution need to be run in a separate, dedicated thread that serves as the communication or progress engine. This thread is automatically bound by the library to a core, and uses some basic heuristics to avoid conflicting with both other processes that may be on the same node and other threads (e.g. OpenMP compute threads) that the application may spawn. Asynchronous operations are submitted to the communication engine as a *state* object that encapsulates the operation to be performed and any necessary state (essentially, a closure). Submission is done via a lock-free single-producer, single-consumer queue (implemented as a classic Lamport queue [38] with modifications described in [39], and could be generalized to a MPSC queue). The engine maintains an internal queue of currently running state objects, and invokes a `step` method on them, which should not block. When the operation has completed, the engine can optionally indicate this to other threads by atomically setting a flag in a request object.

This implementation approach is inspired by the communication engines that have been used in other high-performance communication libraries [40], [41].

Aluminum's MPI backend utilizes the progress engine to provide asynchronous progress on the host both for custom algorithm implementations and via `MPI_Test` polling for non-blocking MPI operations. We do this because we have observed that MPI implementations often do not make adequate progress on their own without polling (see also e.g. [42]). The host-transfer allreduce also makes use of the progress engine to perform communication, as we describe next.

2) Non-blocking and host-transfer allreduce: Aluminum has a heavy focus on non-blocking communication with GPU buffers. For the NCCL backend, non-blocking allreduces are automatically run on one of Aluminum's internal CUDA streams, described in Figure 4. The `Al::Wait` operation implements the synchronization to complete the communication. This allows communication to proceed without blocking the user's stream or the host. In our experiments and profiling, we have observed that this strategy enables excellent communication/computation overlap.

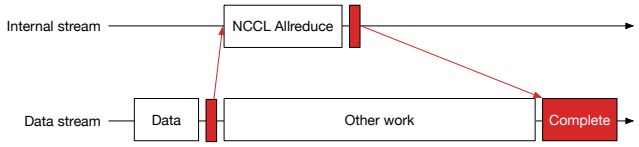


Fig. 4. Performing a non-blocking allreduce using NCCL. Data is computed on a stream by the application, and a separate, internal stream is synchronized to the first. This stream performs the NCCL allreduce, while the data stream can perform other computation. When the result is needed, the internal stream can be synchronized back to the data stream. (Red boxes are synchronization, such as CUDA events. Boxes are not to scale.)

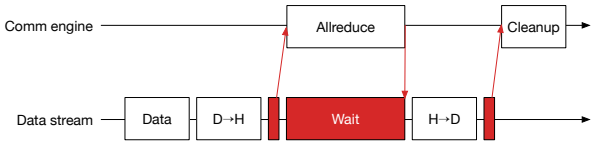


Fig. 5. Implementation of the blocking host-transfer allreduce. The data buffer to transmit is computed on the data stream, after which a device-to-host memcpy transfers moves the buffer to the host. CUDA event synchronization is used to determine when the transfer has completed, after which an MPI allreduce is performed. Meanwhile, the data stream is blocked with a wait operation, until the host signals completion, after which a host-to-device memcpy transfers the buffer back. A second event signals completion of this transfer, so temporary resources can be released. (Boxes are not to scale.)

For latency-dominated workloads, we have implemented a "host-transfer" allreduce that encapsulates MPI's tree-based allreduce algorithms. As described in Section III-B, these can be significantly more performant than NCCL in the right regimes. At a high level, this implementation simply transfers the GPU memory to the host, performs the allreduce in host memory using MPI, and transfers the result back to the GPU. To avoid the caller blocking the host, the operation enqueues the necessary kernels and events on the communicator's stream, and then delegates communication to Aluminum's communication engine. Polling on CUDA events is used to determine when memory transfers have completed. To block the stream while communication is in progress, the `cuStreamWaitValue32` operation from the CUDA driver API is used. This prevents any work submitted to the stream after the call from beginning until a memory location is written. The entire process is described in more detail in Figure 5. A non-blocking version of this is implemented similarly to non-blocking operations for NCCL, by running

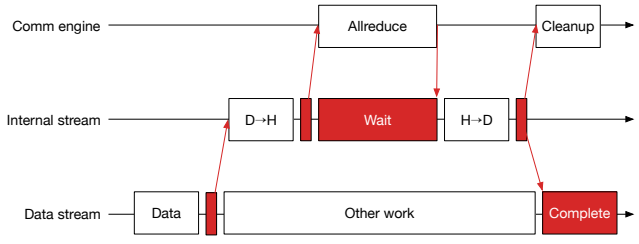


Fig. 6. Implementation of the non-blocking host-transfer allreduce. This is similar to the blocking version, but ron on an internal stream and a separate completion operation is used to invoke the synchronization with the data stream to complete the operation. (Boxes are not to scale.)

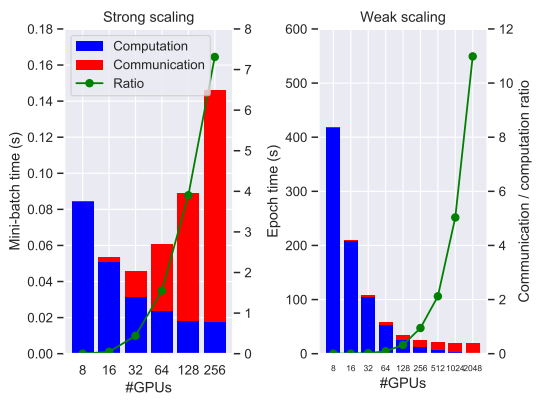


Fig. 7. Strong and weak scaling results for ResNet-50 using our synthetic benchmark on Sierra, using Aluminum+NCCL with communication/computation overlap. The bars break down runtime by computation and unoverlapped communication.

on an internal stream (see Figure 6).

Because we transfer the entire GPU buffer to the host, this approach could be significantly optimized by utilizing GPUDirect RDMA (GDR) [43], and by pipelining for longer messages.

While we have described and implemented this "host-transfer" approach for allreduces, it can be applied to any communication operation. We briefly describe applying this approach to send and recv operations next.

3) Other operations: Send and recv operations that support Aluminum's semantics for GPU buffers are useful both to support applications that require more irregular communication patterns and as building blocks for custom implementations of collectives. Both operations can be implemented similarly to the host-transfer allreduce.

For a send operation, we transfer the data from the GPU to the host and then use `MPI_Isend` within the communication engine to perform the completion. The communicator's stream does not need to be blocked: similarly to MPI's semantics, we consider it locally complete when the user buffer can be reused. For recv, the communication engine can begin an `MPI_Irecv` immediately while blocking the communicator's stream. Once complete, the stream is notified and the buffer transferred to the GPU.

Using these operations as primitives, we have implemented our own ring allreduce in Aluminum's MPI-CUDA backend. This allreduce pipelines communication and host/GPU memory transfers, supports both single- and bi-directional rings, and performs reduction operations on-GPU. While this implementation is not always competitive with NCCL's (in particular, it does not take advantage of GDR), it does enable additional flexibility by supporting reduction operations that NCCL lacks.

VI. Benchmarks

To demonstrate the advantages of our overlapping and latency optimizations, we apply the same benchmark as in II-C, now using Aluminum.

A. Overlapping

Figure 7 plots the runtime and communication/computation breakdown for strong and weak scaling using NCCL with overlap (compare with Figure 2). At small scales, we successfully overlap nearly all communication; indeed, for weak scaling, communication is not a significant factor until 256 GPUs.

For strong scaling the runtimes improves in every case, however, beyond 32 GPUs there is simply too much communication and insufficient computation to hide it. In particular, because many allreduces can only be started toward the end of backprop, allreduces later in backprop always have less computation available to hide them. Nonetheless, overlapping still reduces communication overhead in these cases. 32 GPUs remains the optimal number to use in this case, and runtime is improved by $\sim$1.4x here.

For weak scaling, the constant amount of local computation means that Aluminum is able to hide more of the communication. Unfortunately, at very large scales, communication overheads with NCCL remain too high, and profitability for weak scaling is very low beyond 256 GPUs. At 2048 GPUs, the runtime is almost entirely communication.

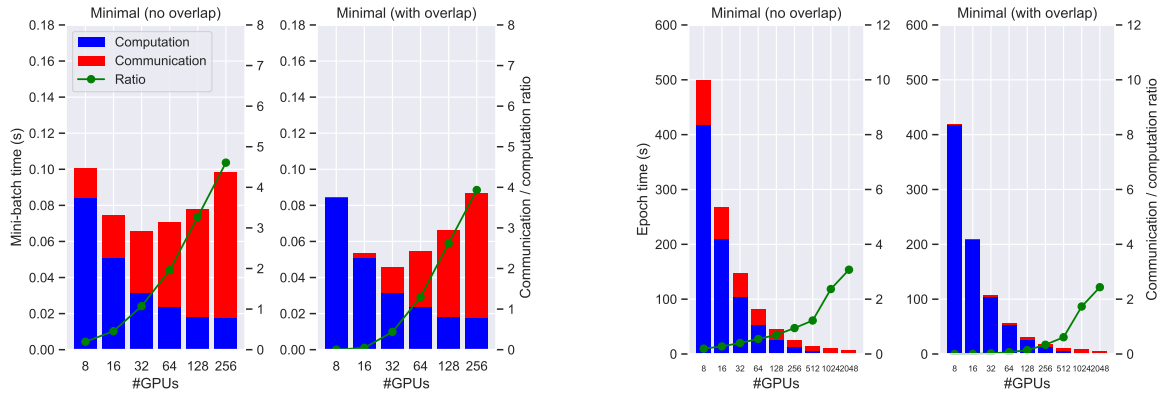
B. Latency

To demonstrate the different regimes in which NCCL and our latency-optimized host-transfer allreduce are better, we conducted a simple benchmark comparing their performance across a range of node/GPU counts (2-512 nodes/8-2048 GPUs) and buffer sizes (1-2^{28} parameters) on Sierra. For each configuration we computed the average over ten runs of the in-place version of allreduce algorithm, after a warmup run. The underlying MPI distribution was MVAPICH2 2.3rc2.

Figure 11 plots the actual performance results for each scale. We can see that NCCL has a significant advantage at the smallest scale (two nodes), that gradually disappears as the number of nodes increases. At small scales, the impact of latency is smaller, so the difference between the ring and tree-based algorithms is relatively small. The host-transfer algorithm starts performing better at 64 GPUs, and at 128 GPUs, it is over 2x faster than NCCL for small messages. At 2048 GPUs, this increases to over 20x. Further, the tree-based allreduces scale much better with increasing node count than NCCL's ring-based allreduce.

Figure 10 plots which implementation is faster for a given configuration, providing a summary of Figure 11. Once running on 64 GPUs (16 nodes), the host-transfer allreduce outperforms NCCL for messages of up to 32768 parameters. At the largest scale, the host-transfer allreduce is preferred for messages up to 2^{19} parameters.

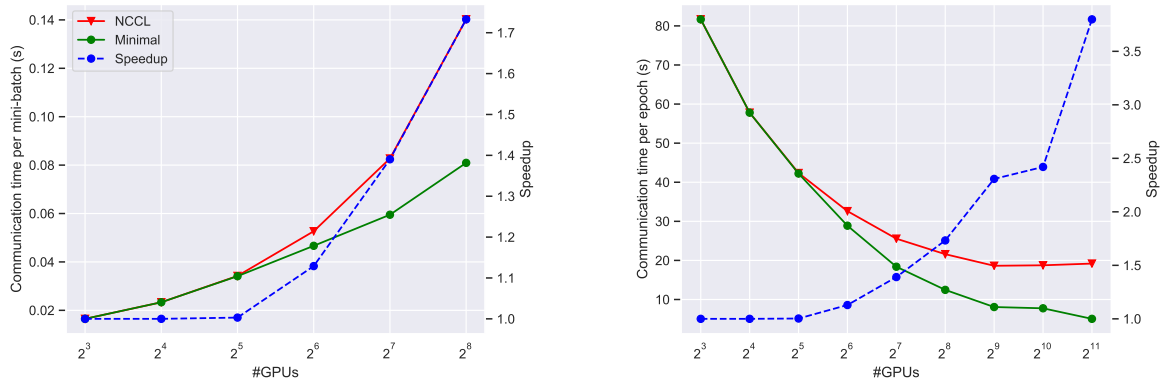
It may be somewhat surprising that NCCL performs well even for very small messages up to 32 GPUs. We attribute this to two factors. First, NCCL is able to take advantage of GPUDirect RDMA [43] and node-local topology information, to reduce communication overhead and latency. Second, our



(a) Strong scaling

(b) Weak scaling

Fig. 8. Strong and weak scaling results for ResNet-50 using our synthetic benchmark on Sierra, using Aluminum to dynamically select either NCCL or our latency-optimized host-transfer allreduce. The bars break down runtime by computation and unoverlapped communication.



(a) Strong scaling

(b) Weak scaling

Fig. 9. Communication time and speedup for strong and weak scaling for ResNet-50 in our synthetic benchmark. The NCCL and minimal lines plot the absolute communication time at that scale, and the speedup line plots the improvement of the minimal algorithm over NCCL at that scale.

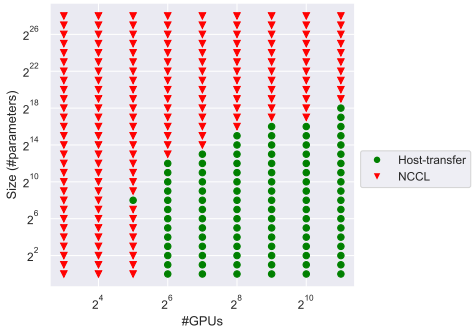


Fig. 10. The fastest allreduce algorithm for a given number of GPUs and buffer size on Sierra. A green dot marks the configurations our host-transfer allreduce is fastest; a red triangle when NCCL is. The host-transfer point for 32 GPUs appears to be due to a protocol change or similar within NCCL.

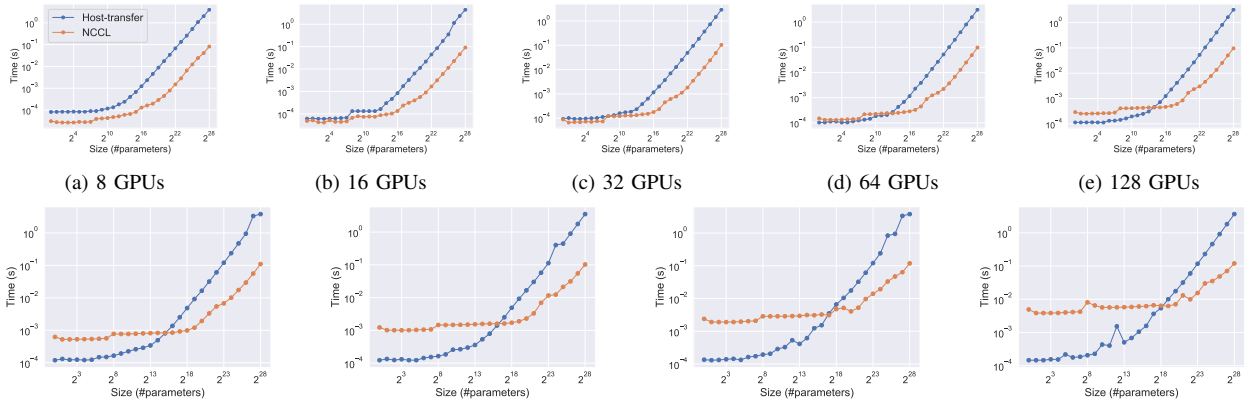
implementation is a prototype whereas NCCL is an optimized production library.

It is important to observe that the size range where the host-transfer allreduce outperforms NCCL corresponds to a significant portion of the allreduces required when training AlexNet or ResNet-50 (see Figure 1). While these allreduces also tend to be faster, improving their performance helps to reduce communication overheads during training.

To this end, we repeat the benchmark from Section II-C with a "minimal" algorithm that is a hybrid of the host-transfer and NCCL allreduces. This algorithm uses our prior benchmarking results to select the fastest implementation for a given input configuration. The results for strong and weak scaling (with and without overlap) are presented in Figures 8a and 8b.

Strong scaling benefits less from the better allreduce algorithms, as the regime where it is profitable is not significantly impacted by them. Nonetheless, at larger scales communication overhead is significantly reduced. This implies that with a better implementation and improved compute scaling, we may be able to successfully strong-scale training further.

Weak scaling exhibits a more noticable impact, dramatically improving the performance at large scales. Whereas NCCL, even with overlap, barely improves performance beyond 256, the minimal algorithm sees continued profit in scaling to 2048



(a) 8 GPUs (b) 16 GPUs (c) 32 GPUs (d) 64 GPUs (e) 128 GPUs

(f) 256 GPUs (g) 512 GPUs (h) 1024 GPUs (i) 2048 GPUs

Fig. 11. Performance results for our host-transfer allreduce and NCCL's allreduce on Sierra.

GPUs. Furthermore, communication overhead, while still quite high, is significantly improved, by over 5x at 2048 GPUs.

To illustrate more directly the communication improvements within the benchmark, we plotted only the communication time for both NCCL and the minimal algorithm in Figure 9. Here we can quite clearly see that speedups in communication begin at 64 GPUs; below that, the minimal algorithm is identical to NCCL. Beyond 64 GPUs, performance improvements accrue rapidly, such that the minimal algorithm is over 4x faster than NCCL alone at 2048 GPUs.

We investigated running multiple allreduces concurrently, but we have observed that NCCL performs only a single allreduce at a time, even if multiple allreduces could be executed. While our host-transfer allreduce does not have this restriction, we did not incorporate this optimization into our benchmarks here.

VII. Training Experiments

To evaluate end-to-end training in a real environment, we integrated Aluminum into the LBANN toolkit [1], which is optimized for training deep networks on large GPU HPC clusters. We train ResNet-50 on the ImageNet-1K dataset [29] using Sierra, with data being read off a Spectrum Scale parallel filesystem.

Strong scaling is performed by fixing the mini-batch size to 256, the default, and increasing the number of GPUs. Weak scaling fixes a per-GPU mini-batch size of 32, increasing the global mini-batch size as the number of GPUs increases. This results in fewer iterations being performed per epoch. Note this is the same setup as in our synthetic benchmark. MVAPICH2 v2.3 was used as the underlying MPI distribution (see Section II-C for other system/software details). Experiments were performed using up to 256 GPUs on 64 nodes[3].

We compare three configurations: LBANN using CUDA-aware MPI, Aluminum with NCCL, and Aluminum with NCCL and the host-transfer allreduce (HT). In the last configuration, a static performance model selects between NCCL and HT, similar to the "minimal" algorithm in our prior benchmarks. Notably, based on our benchmarks, NCCL is preferred exclusively when running on fewer than 64 GPUs. Strong and weak scaling results are presented in Figure 12.

Both strong and weak scaling exhibit similar trends to those in our benchmark (compare with Figure 8). However, I/O is now a major factor in runtime, which was not reflected in it. Other computations (optimization, etc.) are also present. This results in additional work that communication can be overlapped with, reducing overhead.

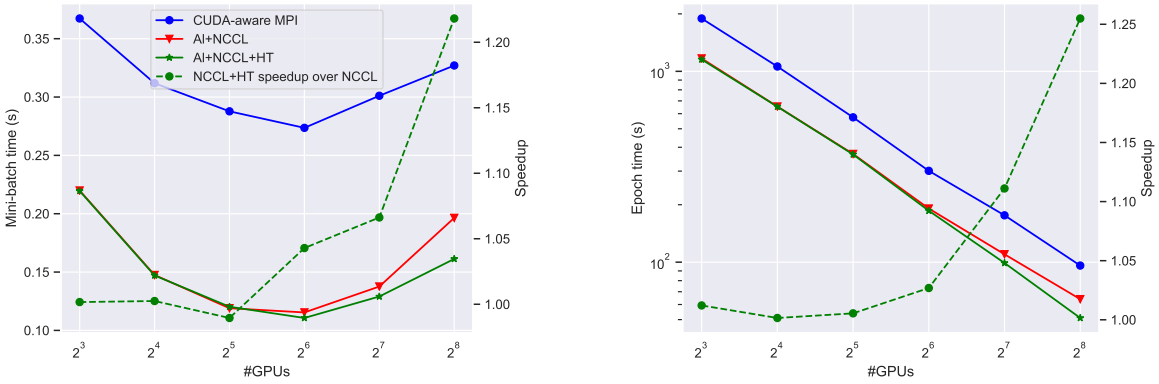
CUDA-aware MPI is significantly outperformed by both Aluminum configurations. Aluminum+NCCL is $\sim$2.4x and $\sim$1.5x faster than CUDA-aware MPI for strong scaling at 64 GPUs and weak scaling at 256 GPUs, respectively. Aluminum+NCCL+HT is $\sim$2.5x and $\sim$1.9x faster in these cases. Aluminum's semantics for communication with GPU buffers means that both NCCL and the host-transfer allreduce are asynchronous with respect to the host, enabling I/O to be overlapped much more extensively.

When strong scaling, this additional work enables scaling to be profitable up to 64 GPUs (compared to 32 GPUs in our benchmark), after which communication overheads and poor compute scaling begin to dominate. At 64 GPUs, the host-transfer algorithm begins to slightly improve communication performance, resulting in a $\sim$1.05x improvement in runtime, which is commensurate with the modest improvements over NCCL our benchmarks show at this scale. We see larger speedups with more GPUs, despite it not being profitable; future communication optimizations may enable strong scaling at these scales.

Weak scaling shows improvements of similar magnitude, except the better compute scaling means that it is profitable up to 256 GPUs. The host-transfer algorithm again shows improvements beginning at 64 GPUs, and results in a $\sim$1.25x performance improvement over NCCL at 256 GPUs.

Overall, both strong and weak scaling demonstrates the advantages of Aluminum over vanilla CUDA-aware MPI, and, at larger numbers of GPUs, the importance of taking latency into consideration when selecting communication algorithms.

[3]We could not scale beyond this due to system issues.



(a) Strong scaling

(b) Weak scaling

Fig. 12. Strong and weak scaling for end-to-end training of ResNet-50 in LBANN with Aluminum on Sierra. (Note the log scale for weak scaling.)

VIII. Related work

Many other frameworks for training deep neural networks, including TensorFlow [3], PyTorch [4], FireCaffe [2], LBANN [1], and CNTK [7] aim to scale training, and optimize communication to that end. While these frameworks often implement a variety of optimizations, they typically rely on either MPI or NCCL to provide the underlying communication layer on dedicated clusters, and therefore can benefit from the optimizations we have discussed and implemented within Aluminum.

There are several communication layers that have been developed primarily to accelerate training deep networks, and can be integrated into existing frameworks such as TensorFlow. These often aim to replace centralized parameter servers with a decentralized allreduce implementation. Baidu's allreduce [5] was the first attempt to leverage ring allreduces for training deep networks, and is implemented atop CUDA-aware MPI to manage GPU communication. Facebook's Gloo [44] supports a number of collective algorithms, including multiple optimized allreduce implementations; it builds upon MPI, node-local NCCL (but not distributed NCCL), and custom communication layers. Uber's Horovod [6] similarly supports allreduces and several other collectives, and builds upon CUDA-aware MPI and NCCL. Horovod supports *tensor fusion*, which attempts to address issues with latency-bound allreduces by merging the buffers together to perform fewer, larger allreduces. Baidu's allreduce implements none of the optimizations we have described; Gloo and Horovod will not block host execution, but do not overlap computation on the GPU, and do not implement latency-optimized allreduces.

NCCL [31] and MPI are most similar in approach to Aluminum, and we build upon both in many ways, as discussed throughout the paper. NCCL lacks native support for non-blocking allreduces and is not latency-optimized; MPI suffers from a semantic mismatch with CUDA that limits its performance. NVSHMEM [45] implements high-optimized point-to-point communication that avoids the semantic issues MPI suffers from while being entirely managed from a GPU. It provides no collective operations, but could be useful as a building block for higher-level systems.

Many works have investigated scaling training by increasing the mini-batch size (weak scaling) [15], [16], [46]. The primary contribution of these approaches works is not the scaling techniques, but the learning techniques used to maintain model accuracy despite the large mini-batch size. [46] does discuss large-scale communication, but similarly to Horovod, addresses latency concerns through tensor fusion.

Another significant body of work has investigated many other approaches to parallelizing training; we refer the reader to [25] for an excellent overview. We view these techniques as orthogonal to our work here: they can be leveraged in concert to improve performance. One important class of work that more closely relates is quantization [19]–[21]. These approaches aim to trade additional local computation to reduce communication volume, and are particularly applicable to optimizing allreduces on large buffers. Quantization complements our work especially well, as it requires good overlap between computation and quantization+communication and shifts more communication into a latency-bound regime. [47] discusses techniques and APIs that enable efficient implementation of quantized communication.

IX. Conclusions

We have examined the communication requirements for training deep neural networks, and found that the overhead of communication is significant, and becomes the dominant cost at scale. We applied several optimizations to reduce this overhead. Overlapping communication and computation and using latency-optimized algorithms helps to directly reduce this overhead. Identifying and working around the semantic mismatch between MPI and CUDA both reduces overheads and enables overlapping of host and GPU computation. We incorporated these improvements into the open-source Aluminum library.

We do not view these optimizations as inherent to Aluminum or the present work, and encourage other libraries, especially MPI distributions, to adopt them. These techniques

are also not limited to training deep networks; other applications that leverage GPUs, such as numerical or graph analytics applications, can benefit from them too. Aluminum's semantics and point-to-point communication implementations are a step toward supporting more irregular communication patterns.

Our work has improved the strong and weak scaling of training deep networks significantly. However, both remain heavily communication-bound at large scales. Strong scaling has always been difficult due to diminishing computational work and increased communication requirements. Aluminum enables profitable weak scaling to large numbers of GPUs, but communication overheads limit the benefits. As GPUs continue to improve computational performance, while network bandwidth grows slowly and latencies reach physical limits, communication will only become a more critical bottleneck in the future. To efficiently utilize large GPU clusters, implementors must pay close attention to the optimizations described here, and develop improved techniques that reduce latencies, minimize overheads, and ultimately scale communication.

Acknowledgments

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-757866). Funding provided by LDRD #17-SI-003. Additionally, some of the testing and development support work needed to complete this research was funded by the Sierra Institutional Center of Excellence at LLNL. Experiments were performed at the Livermore Computing facility. The authors would like to thank the LBANN team for their assistance.

References

[1] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "LBANN: Livermore Big Artificial Neural Network HPC toolkit," in *MLHPC*. ACM, 2015, p. 5.

[2] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: near-linear acceleration of deep neural network training on compute clusters," in *CVPR*, 2016, pp. 2592–2600.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017.

[5] Baidu Research, "Baidu allreduce," https://github.com/baidu-research/baidu-allreduce, 2018.

[6] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[7] F. Seide and A. Agarwal, "CNTK: Microsoft's open-source deep-learning toolkit," in *KDD*. ACM, 2016, pp. 2135–2135.

[8] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *ICML*, 2013, pp. 1337–1345.

[9] M. D. Schatz, R. A. Van de Geijn, and J. Poulson, "Parallel matrix multiplication: A systematic journey," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C748–C781, 2016.

[10] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[11] S. Hochreiter and J. Schmidhuber, "Flat minima," *Neural Computation*, vol. 9, no. 1, pp. 1–42, 1997.

[12] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[13] P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, C. Baldassi, C. Borgs, J. Chayes, L. Sagun, and R. Zecchina, "Entropy-SGD: Biasing gradient descent into wide valleys," *arXiv preprint arXiv:1611.01838*, 2016.

[14] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, "Sharp minima can generalize for deep nets," *arXiv preprint arXiv:1703.04933*, 2017.

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

[16] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," *CoRR, abs/1709.05011*, 2017.

[17] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech DNNs," in *ICASSP*. IEEE, 2014, pp. 235–239.

[18] J. Keuper and F.-J. Preundt, "Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability," in *MLHPC*. IEEE Press, 2016, pp. 19–26.

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

[20] N. Dryden, T. Moon, S. A. Jacobs, and B. Van Essen, "Communication quantization for data-parallel training of deep neural networks," in *MLHPC*. IEEE, 2016, pp. 1–8.

[21] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *NIPS*, 2017, pp. 1709–1720.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[24] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.

[25] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *arXiv preprint arXiv:1802.09941*, 2018.

[26] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.

[28] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "DAWNBench: An end-to-end deep learning benchmark and competition," in *NIPS*, 2017.

[29] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[30] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[31] NVIDIA, "NVIDIA collective communications library," https://developer.nvidia.com/nccl, 2018.

[32] Lawrence Livermore National Laboratory, "Sierra," https://hpc.llnl.gov/hardware/platforms/sierra, 2018.

[33] S. Ioffe, "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models," in *NIPS*, 2017, pp. 1945–1953.

[34] Y. Wu and K. He, "Group normalization," *arXiv preprint arXiv:1803.08494*, 2018.

[35] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.

[36] Hydrogen team, "Hydrogen," https://github.com/LLNL/Elemental, 2018.

[37] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM TOMS*, vol. 39, no. 2, p. 13, 2013.

[38] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.

[39] N. M. Lê, A. Guatto, A. Cohen, and A. Pop, "Correct and efficient bounded FIFO queues," in *SBAC-PAD*. IEEE, 2013, pp. 144–151.

[40] A. Brooks, H.-V. Dang, N. Dryden, and M. Snir, "PPL: an abstract runtime system for hybrid parallel programming," in *ESPM2*. ACM, 2015, pp. 2–9.

[41] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *EuroMPI*. ACM, 2016, pp. 1–14.
[42] P. R. Eller and W. Gropp, "Scalable non-blocking preconditioned conjugate gradient methods," in *Supercomputing*. IEEE Press, 2016, p. 18.
[43] NVIDIA, "GPUDirect RDMA," https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2018.
[44] Facebook, "Gloo," https://github.com/facebookincubator/gloo, 2018.
[45] S. Potluri, A. Goswami, D. Rossetti, C. Newburn, M. G. Venkata, and N. Imam, "GPU-centric communication on NVIDIA GPU clusters with InfiniBand: A case study with OpenSHMEM," in *HiPC*. IEEE, 2017, pp. 253–262.
[46] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.
[47] C. Renggli, D. Alistarh, and T. Hoefler, "SparCML: High-performance sparse communication for machine learning," *arXiv preprint arXiv:1802.08021*, 2018.