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  
†Department of Computer Science  
University of Illinois at Urbana-Champaign

Abstract—We identify communication as a major bottleneck for training deep neural networks on large-scale GPU clusters, taking up to 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 training algorithm’s mini-batch size, 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 each node is 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 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. We propose instead to make our communication operations aware of the stream a GPU datum is being computed on. This enables the communication operation to function similarly to a CUDA kernel, and minimizes the synchronization overheads without impacting pipelining. 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 that getting good performance for fine-grained, often latency-dominated communication as critical. 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 that can be used for performing 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 that implements our communication techniques.
- We evaluate the impact of these methods in both microbenchmarks and end-to-end training within the open-source LBANN toolkit.

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, and is 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 (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 that need to be learned, 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 processor’s 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 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

1URL removed for anonymization.

2One could equivalently think of communication as being performed during the optimization phase; we choose backprop for convenience.
Fig. 1. Histograms breaking down the number of parameters in each parameter buffer (essentially, a layer) for the AlexNet and ResNet-50 networks. In our implementation, each parameter is a 4-byte float.

(a) AlexNet

(b) ResNet-50

Fig. 2. Strong scaling results for ResNet-50 using our benchmark on Sierra, with either NCCL or a dynamic minimal-time algorithm, and with and without overlap of communication/computation. The bars show the total time for a mini-batch, broken down by computation and unoverlapped communication. Superimposed is the ratio of communication to computation. Note that each Sierra node has four GPUs.

(a) NCCL

(b) Minimal

Fig. 3. Weak scaling results for ResNet-50 using our benchmark on Sierra, with either NCCL or a dynamic minimal-time algorithm, and with and without overlap of communication/computation. The mini-batch size per GPU is fixed to 32, so the global mini-batch size increases with scale. Since the total dataset size is fixed, more GPUs results in fewer, larger iterations. The bars show the total time for an epoch, broken down by computation and unoverlapped communication. Superimposed is the ratio of communication to computation. Note that each Sierra node has four GPUs.
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 for its scale parameters and one for its bias parameters.

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 across this entire regime, as we discuss in Section III.

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 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 2.2.13.

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 2a (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 3a (left). In this case, computation scales linearly. The total time for communication decreases as the number of GPUs increases, because fewer iterations are performed, resulting 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 10x ratio of communication to computation on 1024 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. OVERLAP AND LATENCY

We now discuss two basic optimizations for reducing communication overhead and improving performance: overlapping and latency-efficient allreduces algorithms. Neither of these techniques are new. Overlapping communication and computation during training has been discussed before (e.g. [15]), and we 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 concurrently performing allreduces to compute global gradient updates and backpropagation or optimization for other layers. This can be done within the constraints discussed in Section II-A. Thus, to maximize the potential for overlapping, within each layer gradients should be computed first, and then an asynchronous allreduce started 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 utilizing the NCCL library requires additional work. The communication operations are enqueued as kernels on a CUDA stream, hence if they are enqueued on the same stream as training computations are performed on the operations will block. Instead, the NCCL allreduce should be performed on a separate CUDA stream, and appropriate synchronization added. We illustrate this strategy in more detail in Figure 4. In our experiments and profiling, we have observed that this strategy enables excellent communication/computation overlap.

Using this method, we perform the same benchmarks as in II-C. Figures 2a and 3a demonstrate the improvements. At small scales for both strong and weak scaling, the bulk of communication is successfully hidden. This further results in significantly improved communication/computation ratios. At larger numbers of GPUs (32 and above for strong scaling, 128 and above for weak scaling), there is insufficient computation to hide the communication at that scale. 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.

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.

For data lying in host memory, it is easy to rely on optimized MPI distributions, which provide a suite of allreduce algorithms tuned to these different regimes [35]. As our data of interest lies exclusively in GPU memory, we could make use of a CUDA-aware MPI runtime; however, as we discuss in Section IV, directly utilizing MPI results in overheads because of MPI’s interaction with CUDA. We instead make use of our own implementation (discussed later), “host-transfer”, that wraps around MPI and is preferred for small messages at larger scales.

This enables our allreduces to take advantage of tree-based algorithms for latency-bound workloads and large numbers of processes. These can be algorithmically better than ring algorithms in many regimes. In particular, we utilize recursive-doubling and recursive-halving/recursive-doubling algorithms. Recursive-doubling is preferred for small messages and has optimal latency. The recursive-halving/recursive-doubling algorithm (also called Rabenseifner’s algorithm) has slightly
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 2b and 3b.

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 noticeable impact, dramatically improving the performance at large scales. Whereas NCCL, even with overlap, barely improves performance from 256 to 1024 GPUs, the minimal algorithm sees continued profit in scaling to 1024 GPUs. Furthermore, communication overhead—while still quite high—is significantly improved, by up to \( \sim 2.6x \) at 1024 GPUs.

An additional optimization, not reflected within these benchmarks, 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. One limitation that we have observed with NCCL is that it performs only a single allreduce at a time, even if multiple allreduces could be executed. Our host-transfer allreduce implementation does not have this restriction.

IV. INTERFACING WITH MPI

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, when a kernel begins execution, all its inputs are ready. Kernel launches (along with most other CUDA operations) are asynchronous and do not block the host, but there is a cost (roughly 10 \( \mu s \)) 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, and prevents pipelining kernel launches to hide their launch latency or overlapping 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). As shown in Figures 2 and 3, overlapping is an important performance optimization. Blocking the host to wait for operations to complete also limits its ability to overlap other operations, such as I/O, with computation and/or communication. In the context of training deep nets, I/O can be quite expensive, so hiding it is crucial. Finally, we have shown that latency-optimized communication is important for scaling; the additional synchronization imposed will increase latency.

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 comes from the idea that we should treat MPI communication operations as “just another kernel” to be enqueued on a stream. A proof-of-concept that communication can be treated this way is NCCL, where each operation takes a stream as an argument and results in the usual semantics for a kernel launch: 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 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. The details associated with achieving the remaining characteristics are somewhat complicated and implementation-dependent. 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 should be 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. This library encapsulates the optimizations discussed throughout this paper, such as easy non-blocking operations on both host and GPU (Section III-A), latency-optimized algorithms
**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 API to invoke a non-blocking, in-place allreduce (for example) looks like: $\text{Al::NonblockingAllreduce<Backend>}(\text{buffer}, \text{count}, \text{op}, \text{comm}, \text{req})$. where $\text{buffer}$ and $\text{count}$ define the buffer to be reduced, $\text{op}$ is a reduction operation (e.g. summation), $\text{comm}$ is an Aluminum communicator object, and $\text{req}$ 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: $\text{Al::Wait<Backend>}(\text{req})$. 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 $\text{Al::Wait}$ operation were used with the MPI-CUDA backend, it would (perhaps counterintuitively) not block the host, but instead block $\text{comm}$’s CUDA stream until the allreduce completed.

### B. Some 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.

![Diagram of Aluminum and its interfaces](image-url)

**Fig. 9.** Integration of Aluminum into the open-source toolkits, LBANN and Hydrogen.
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 [39] with modifications described in [40], 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 [41], [42].

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. [43]). 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. Non-blocking operations via the NCCL backend are automatically run on one of Aluminum’s internal CUDA streams, as discussed in Section III-A and Figure 4. The Al::Wait operation implements the synchronization to complete the communication. All of this is done with asynchronous CUDA operations and does not block the caller.

For latency-dominated workloads, we have implemented a “host-transfer” allreduce that, as described in Section III-B, 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 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 are 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 7. A non-blocking version of this is implemented similarly to non-blocking operations for NCCL, by running on an internal stream (see Figure 8).

Because we transfer the entire GPU buffer to the host, this approach could be significantly optimized by utilizing GPUDirect RDMA [36], 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_Recv 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 GPUDirect RDMA), it does enable additional flexibility by supporting reduction operations that NCCL lacks.

VI. TRAINING EXPERIMENTS

Note: Potentially temporary results pending NCCL bugfix (or I will make these final and expand).

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

Results comparing Aluminum’s NCCL backend with CUDA-aware MPI (MVAPICH2 version 2.3) are presented in Figure 10, where we observe that Aluminum+NCCL significantly outperforms CUDA-aware MPI. This improvement is primarily a result of using Aluminum’s synchronization semantics, which not only enables communication/computation overlap, but avoids blocking the host or repeatedly paying kernel launch overheads. A major advantage of not blocking the host is that I/O and data preprocessing for a subsequent mini-batch is completely overlapped with the backprop and optimization phases of the prior mini-batch.

VII. 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 large 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 allreduce 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 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-aware MPI took too long to run.)

VIII. 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, while developing improved techniques to reduce latencies and other overheads and scale communication.

ACKNOWLEDGMENTS

Prepared by LLNL under Contract DE-AC52-07NA27344. Funding provided by LDRD #17-SI-003. Experiments were performed at the Livermore Computing facility. The authors would like to thank the LBANN team for their assistance.

REFERENCES


