Scalable Machine Learning with OpenSHMEM

Gerard Taylor, David Ozog, Md. Wasi-ur- Rahman, and James Dinan

Intel Corporation

Abstract—**Deep convolutional neural networks (DNNs) have had a significant and lasting impact across the computing industry. Training these large neural networks is computationally intensive and is often parallelized to shorten training times that could otherwise range from days to weeks. The Message Passing Interface (MPI) communication model has been commonly used to facilitate the data exchange and synchronization required for parallel DNN training. We observe that OpenSHMEM supports many of the same communication operations as MPI — in particular, the all-reduce operation needed to support data parallelism and that OpenSHMEM may further provide a unique solution to fine-grain model parallel computation. In this work, we present an initial evaluation of OpenSHMEM's suitability for use in DNN training and compare the performance with MPI. Results indicate that OpenSHMEM data parallel performance is capable of achieving a 30% decrease in training time when compared to MPI. The usage of OpenSHMEM to support model parallelism will be explored in our future work.**

Index Terms—**Deep Learning, OpenSHMEM, PGAS, parallel computing**

I. Introduction

Over the last decade, advances in machine learning and specifically deep learning have had a lasting impact on the world. Deep learning methods have produced incredible results in the fields of natural language processing, computer vision, and robotics to name a few [8]. However, these results come at a cost. In order to train a Deep Neural Network (DNN) for a particular task with a certain level of accuracy, the neural networks must be incredibly large and be trained on massive datasets. As a result of the increased depth and complexity of these models, training time can be on the order of weeks to months. This caveat has led researchers to look for methods to parallelize training to reduce such long training times.

Parallelization of the DNN synchronous Stochastic Gradient Descent (SGD) training algorithm typically targets either data or model parallelism [3]. Data parallelism partitions the mini batch of training data points in a given epoch into N equally sized chunks and N copies of the model are trained in parallel on these chunks. The resulting gradients are then averaged and broadcast using a collective all-reduce communication operation and all N copies of the model receive the same updates before progressing to the next training epoch. This process is repeated until the model converges to the desired accuracy. Model parallelism divides the work involved in the forward propagation calculation on a single training data point across multiple computational resources. In contrast with the periodic, global data exchange involved in data parallelism, model parallel data exchange is more fine-grained and the overheads must be carefully managed in order to achieve speedup. Data and model parallelism are complementary and, when combined, the technique is referred to as hybrid parallelism.

There are various communication models that are popular amongst the deep learning researchers with the most popular being the Message Passing Interface (MPI). MPI is the leading programming model for High Performance Computing (HPC) applications and, because of the similarities between Deep Learning and HPC, MPI has also become a widely used communication model for parallel DNN training on distributed clusters. OpenSHMEM is also gaining popularity within the HPC community as a Partitioned Global Address Space (PGAS) alternative to MPI. The OpenSHMEM communication model provides one-sided semantics and has been shown to enable higher throughput and lower overheads for several workloads. OpenSHMEM supports the global all-reduce operations needed to support data parallelism. In addition, its global address space style of computation is lightweight and may provide a new method for enabling efficient model parallelism.

While the usage of MPI has been well studied in the context of scale out deep learning, to our knowledge, the suitability of OpenSHMEM for this task has not yet been evaluated. In this work, we present the first evaluation of OpenSHMEM as a communication substrate for scaling deep learning workloads. We use the Intel® Machine Learning Scaling Library (MLSL) [7] to provide an abstraction for the data management in DNN training. MLSL provides an MPI communication back-end, which we extend to also support OpenSHMEM, enabling direct comparisons between MPI and OpenSHMEM performance characteristics. Early results indicate higher parallel efficiency for data parallel scaling when using OpenSHMEM communication versus MPI. When taken in conjunction with the potential suitability of OpenSHMEM to enable model parallelism under a single parallel programming system, this result suggests that deeper exploration is needed to further evaluate OpenSHMEM as an alternative model for scaling deep learning.

II. Background

This paper introduces OpenSHMEM and discusses its use as an alternative to MPI for the underlying communication used in distributed DNN training. This section provides background information regarding the OpenSHMEM programming model, the Intel® Machine Learning Scaling Library [7], and the Intel® distribution of Caffe [4].

A. *OpenSHMEM*

OpenSHMEM [6] is a Partitioned Global Address Space (PGAS) programming model that allows for single-sided communication on distributed compute clusters. The PGAS programming model emphasizes a process's ability to perform Remote Direct Memory Access (RDMA) operations without needing to interrupt the target process. This model differs from the classic MPI model, which involves matching send and receive operations to facilitate communication between remote processes. We note that the MPI Remote Memory Access (RMA) interfaces also support single-sided communication, with capabilities similar to OpenSHMEM. This initial work focuses on collective communication and, therefore, primarily examines performance differences between OpenSHMEM and MPI collective operations. Like MPI, OpenSHMEM programs follow a Single Program, Multiple Data (SPMD) model. Unlike MPI, OpenSHMEM designate a special region of symmetric memory on which single-sided communication operations occur.

Both OpenSHMEM and MPI support the collective routines that are the most pertinent to distributed deep learning - broadcasts and reductions. However, OpenSHMEM collectives inherently operate on the symmetric memory regions, which presents opportunities for efficiency that are not as readily apparent as in MPI collectives. Collective operations are typically implemented using the point-to-point communication operations provided by the base programming model. Thus, OpenSHMEM collectives are often implemented using one-sided communication, while MPI collectives are implemented using two-sided communication. In the context of data parallel deep learning, each processing element (PE) creates its own copy of the neural network model and operates on a subset of the training data. Then, built-in collective communication routines would be used to synchronize the model parameters (weights, biases, and gradients) across all PEs.

B. *Intel® Machine Learning Scaling Library*

The Intel® Machine Learning Scaling Library (MLSL) [7] seeks to efficiently implement common collective communication patterns that are seen in multi-node distributed deep learning training. This library has an easy-to-use API that can be plugged into common deep learning frameworks such as Caffe [4], TensorFlow [2], and many others to accelerate DNN training on multiple nodes. The production version of MLSL uses MPI as its underlying communication protocol. In this work, we modified MLSL to use OpenSHMEM as its communication protocol.

Data parallel training of a DNN typically consists of synchronization of location (barriers), parameters (reductions / broadcasts), and test results (reductions). MLSL implements wrappers to these MPI API functions in such a way as to ensure that the synchronization efforts are efficient by leveraging threads and special cases such as on-node communications which do not have to be sent over the fabric.

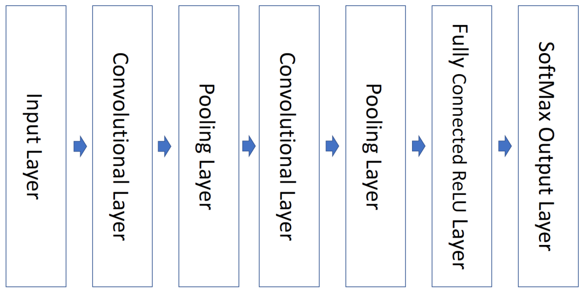

Fig. 1. LeNet DCNN Architecture

C. *Intel Distribution of Caffe*

The Caffe* [4] deep learning framework is targeted toward efficient image processing and was developed by the Berkeley Vision and Learning Center (BVLC). While Caffe is primarily used for computer vision learning tasks, it is modular enough to be adapted to other learning tasks such as Automatic Speech Recognition, and Reinforcement Learning.

Out of the box, Caffe comes with two modes: CPU and GPU. The CPU only mode executes training and testing of the model on just the CPU. Whereas, the GPU mode seeks to offload some of the computationally intensive tasks such as matrix multiplication to the GPU and thereby accelerate training time. Caffe is not equipped with multi-node execution functionality.

The Intel® Optimization for Caffe* (Intel-Caffe) extends the basic version of Caffe to include multi-node functionality. This multi-node functionality is provided by MLSL, which uses MPI for as its communication protocol. Additionally, Intel-Caffe adds the option to accelerate some of the DNN math computation by leveraging the Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN). Because MLSL has a common external API, changing the underlying communication protocol had a minimal impact on the flexibility of the framework and its ability to interface with Intel-Caffe.

III. Design and Implementation

Caffe is equipped with a set of examples that allow a user to easily get started with the framework. In this work, we train a model to recognize handwritten digits using the LeNet Solver that ships with Caffe. The MNIST [5] handwritten digit data set is comprised of 60,000 training examples and 10,000 test examples. Each example is a 28x28 pixel image of a handwritten number from 0-9.

The LeNet Solver network, referenced in Figure 1, is a relatively small deep neural network and consists of two convolution layers, each of which is followed by a pooling layer. The final pooling layer feeds into a fully connected layer with a Rectified Linear Unit (ReLU) activation function which in turn feeds into the output layer with a SoftMax activation function that is typically used for classification tasks.

Our experiment consisted of training the LeNet Solver network to recognize handwritten digits using Intel-Caffe using

TABLE I
MPI vs. OpenSHMEM Enabled DCNN Training Time

Nodes	PPN	Time (Sec) - MPI	Time (Sec) - SHMEM
1	1	**44.87**	49.64
1	2	62.14	**45.04**
2	1	48.12	**48.04**
2	2	66.45	**47.96**
4	1	51.22	**50.39**
4	2	71.34	**45.28**
8	2	72.61	**47.84**

TABLE II
MPI vs. OpenSHMEM Enabled DCNN Training Accuracy

Nodes	PPN	Accuracy - MPI	Accuracy - SHMEM
1	2	0.9904	0.9868
2	1	0.9909	0.9861
2	2	0.9906	0.9823
4	1	0.9906	0.9820
4	2	0.9913	0.9755
8	2	0.9901	0.9626

MPI enabled MLSL, and Intel-Caffe using OpenSHMEM enabled MLSL. The experiments were conducted on the National Energy Research Scientific Computing Center's (NERSC) Cori [1] System using 16-core Intel® Xeon™Processor E5-2698 v3 ("Haswell") at 2.3 GHz nodes with Cray* MPICH 7.7.6, Cray* SHMEM 7.7.6 on the Aries network.

IV. Evaluation

Preliminary results shown in Table I show that MPI provides the best single PE (i.e., process) training time, but that OpenSHMEM enabled MLSL yields significantly better multi-PE training time (bold entries). We observe that the LeNet model used in this experiment is a relatively small problem with limited ability to take advantage of scaling, as there is little performance improvement from increasing the number of PEs. As a result, the strong scaling data shown in Table I represents a challenging problem that highlights the parallel efficiency achieved by OpenSHMEM versus MPI. For example, at higher numbers of PEs, we observe that MPI slows down relative to one PE, while OpenSHMEM achieves better scaling behavior yielding up to a 30% improvement over MPI. As the number of PEs per node (PPN) is increased, we observe that OpenSHMEM performance improves, whereas MPI performance degrades. This is a result of OpenSHMEM's improved ability to leverage shared memory communication between PEs located on the same node. The resulting accuracy of the trained model is shown in Table II. In both cases, the model converges to a high degree of accuracy. We attribute the differences in accuracy to floating point rounding differences in different implementations of the reduction operation.

One particular design alternative that can lead to further performance impact would be the selection of different root process for the broadcast operations. Broadcasts in the OpenSHMEM API designate a root PE that sends the contents of its parameter buffer to the destination parameter buffers of all other remote PEs. There are several methods of choosing the root PE in deep learning applications. By default, the framework chooses the process with rank 0 to perform the broadcast. However, the root process could be the PE that holds the neural network model that had the best performance (lowest error) or the root PE could be kept constant and all of the model parameters could be averaged and then broadcast to all PEs (the default behavior).

Both methods would require additional collective reductions. If we decide to use the performance based reduction, we would first make a call to the min reduction collective which would return the minimum error across all models. Then we could conditionally broadcast the parameters of the PE that had the lowest error. If instead we chose to average the parameters, we would make a call to the sum reduction collective passing in the parameter buffer and divide the contents of the destination buffer by the number of active PEs which would be followed by a broadcast from a fixed PE. In the future, we plan to investigate more on this direction.

V. Conclusion

This work provides an initial evaluation of OpenSHMEM for data parallel scaling of deep convolutional neural network training. We demonstrated a solution that extends the Intel® Machine Learning Scaling Library to enable a direct comparison between MPI and OpenSHMEM scaling performance using the Intel® distribution of Caffe. We conducted a strong scaling study using the relatively small LeNet handwritten digit classifier that highlighted the improved scaling efficiency of OpenSHMEM, yielding a 30% improvement in training time versus MPI at the scaling limit.

Because OpenSHMEM provides a global address space programming model, it has the potential to address not only data parallel scaling, but also model parallel scaling. In future work, we will investigate OpenSHMEM's ability to support the fine-grain data exchange and synchronization required to efficiently address this inner level of parallelism in DNN training.

References

[1] Cori. http://www.nersc.gov/users/computational-systems/cori/.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[5] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[6] OpenSHMEM Application Programming Interface, Version 1.4. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf, December 2017.

[7] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj Kalamkar, Dipankar Das, Mikhail E. Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, and Pradeep Dubey. On scale-out deep learning training for cloud and hpc, 2018.

[8] Torsten Hoefler Tal Ben-Nun. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. 2018.