MagmaDNN: Towards High-Performance Data Analytics and Machine Learning for Data-Driven Scientific Computing

Daniel Nichols[1], Nathalie-Sofia Tomov[1], Frank Betancourt[1], Stanimire Tomov[1], Kwai Wong[1], and Jack Dongarra[1,2]

[1] University of Tennessee, Knoxville TN 37996, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
{dnicho22, ntomov, fbetanco}@vols.utk.edu, tomov@icl.utk.edu, kwong@utk.edu, dongarra@icl.utk.edu

Abstract. In this paper, we present work towards the development of a new data analytics and machine learning (ML) framework, called MagmaDNN. Our main goal is to provide scalable, high-performance data analytics and ML solutions for scientific applications running on current and upcoming heterogeneous many-core GPU-accelerated architectures. To this end, since many of the functionalities needed are based on standard linear algebra (LA) routines, we designed MagmaDNN to derive its performance power from the MAGMA library. The close integration provides the fundamental (scalable high-performance) LA routines available in MAGMA as a backend to MagmaDNN. We present some design issues for performance and scalability that are specific to ML using Deep Neural Networks (DNN), as well as the MagmaDNN designs towards overcoming them. In particular, MagmaDNN uses well established HPC techniques from the area of dense LA, including task-based parallelization, DAG representations, scheduling, mixed-precision algorithms, asynchronous solvers, and autotuned hyperparameter optimization. We illustrate these techniques and their incorporation and use to outperform other frameworks, currently available.

Keywords: Machine learning · high-performance DNN · data-driven scientific computing

1 Introduction

Powered by hardware advances and availability of massive training data, data analytics and machine learning (ML) research, e.g., using Deep Neural Networks (DNN), have exploded in recent years, making major contributions in applications of computer vision, speech recognition, robotics, natural language processing, and many others. Many of these are scientific applications, where accelerating the DNN training is a major challenge and a current main bottleneck to scale the computation on current and up-coming architectures. In this paper, we present a new data analytics and machine learning framework, called MagmaDNN. Our main goal is to provide scalable, high-performance data analytics

and ML solutions for scientific applications running on current, as well as upcoming heterogeneous many-core GPU-accelerated architectures. To this end, as much of the functionalities needed are based on standard linear algebra routines, we designed MagmaDNN to derive its power from the MAGMA library [15]. The close integration provides the fundamental (scalable high-performance) linear algebra routines available in MAGMA as a backend to MagmaDNN. We present some design issues for performance and scalability that are specific to machine learning (ML) using Deep Neural Networks (DNN), as well as the MagmaDNN designs towards overcoming them. In particular, MagmaDNN uses well established HPC techniques from the area of dense linear algebra, including task-based parallelization, DAG representations, scheduling, mixed-precision algorithms, asynchronous solvers, and autotuning. We illustrate these techniques and their incorporation and use to outperform other frameworks, currently available.

2 MagmaDNN design

Many ML and data analytics problems can be cast as linear algebra (LA) problems, and therefore can be accelerated with familiar algorithms, e.g., BLAS, linear solvers, eigensolvers, or singular value decomposition (SVD), that are routinely used in HPC. These LA algorithms are readily available in highly optimized numerical LA libraries on new architectures, like MAGMA, which is used by MagmaDNN (see the MagmaDNN software stack illustrated on Figure 1, Left). Figure 1, Right shows that significant acceleration can be achieved by using HPC library like MAGMA, e.g., in this case on SVD for square matrices in double precision on two 10 core Intel Haswell E5-2650 v3 CPUs with an NVIDIA V100 GPU accelerator [7].

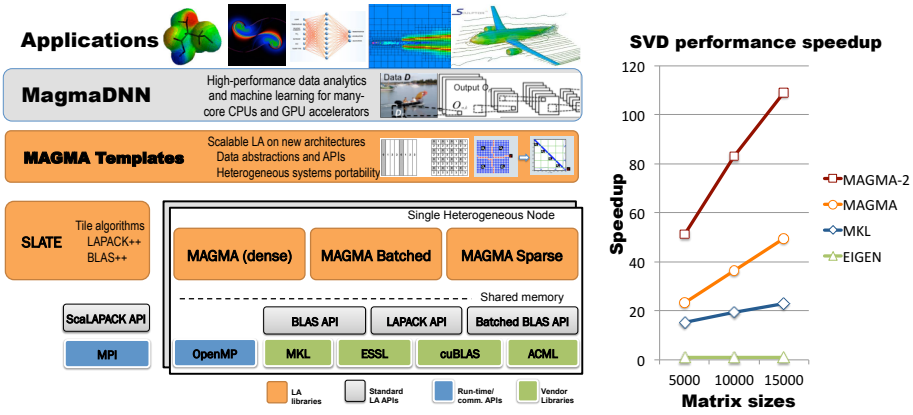

Fig. 1. Left: MagmaDNN software stack. **Right:** MAGMA backend speedup in accelerating fundamental data analytics kernels, e.g., SVD.

Related to DNNs, MagmaDNN's design is modular with each component built on top of each other in increasing levels of abstraction. The core component is the **MemoryManager** (2.1), which handles the underlying memory operations. On top of the MemoryManager, sits 3 other components: **Tensor** (2.2), **Layer** (2.3), and **Model** (2.4). All of these compose the typical MagmaDNN program workflow (2.5).

2.1 MemoryManager

Typical deep learning frameworks hide memory from the user in a Python interface. MagmaDNN accomplishes similar abstraction through its MemoryManager class, which defines and controls memory movement in the framework. Memory is broken into four types: *HOST*, *DEVICE*, *MANAGED*, and *CUDA_MANAGED*. The latter two are both managed style memory, which keep track of data on both the CPU and GPU. *CUDA_MANAGED* uses CUDA's unified memory. Similar to unified memory *MANAGED* keeps track of CPU and GPU data, however it must be explicitly synchronized.

Memory bugs are common in GPU and especially C/C++ code. The MemoryManager solves this development hurdle for the library user, while still giving explicit control over memory transactions.

In addition to providing a simple modular interface the MemoryManager provides a flexible platform for MagmaDNN to optimize memory tasks in training. In deep learning data sets are typically large and create a memory bandwidth bottleneck. Several tricks are employed in MagmaDNN such as asynchronous data prefetching and custom synchronization scheduling. Additionally, the customizable nature of the class allows for future work in memory optimization.

2.2 Tensor

At the core of most deep learning frameworks is the tensor: a structure storing multi-dimensional data. In addition to the tensor itself, essential to deep learning is also a collection of math functions, which operate on tensors. MagmaDNN defines many tensor operations using a combination of its own definitions, MAGMA, and CuDNN [6]. The close relation to linear algebra exposes the opportunity to use high performance LA packages, such as MAGMA, to use both the multi-core CPU and GPU devices [15]. Other operations are implemented using optimized CUDA kernels.

The tensor implementation in MagmaDNN wraps around the MemoryManager and gives structure to the linear memory. It aims to provide a simple interface for tensor interaction in addition to Pythonic style indexing.

Modern networks are more than just linear transformations and are typically composed of convolutional and recurrent layers. MagmaDNN provides convolutional support through Batched GEMMs [2], the Winograd algorithm [5], and FFTs [16,13], but for current performance defaults to CuDNN [6].

2.3 Layer

MagmaDNN provides a simple Layer interface for creating network layers. Each layer keeps track of its own forward/backward pass function, weight tensor and bias tensor. Currently MagmaDNN provides seven layer types: *Input*, *Fully Connected*, *Activation*, *Conv2D*, *Pooling2D*, *Dropout*, and *Output*.

Despite all layers inheriting from a base Layer class they each define their own set of parameters. The activation layer accepts `tanh`, `sigmoid`, and `relu` as activation functions and it is simple to define new ones. By all inheriting from the same Layer superclass it is possible to define custom layers for use in training.

2.4 Model

MagmaDNN's Model class defines a typical training routine. This routine will load new data, forward propagate, backward propagate, and update the network. It calculates and stores simple training metrics such as loss, accuracy, and training time.

Typical DNN framework users are not concerned with the specifics of network training routines. Models removes the necessity for end-users to implement their own training loop.

2.5 Workflow

Figure 2 illustrates the typical kernels that are needed in a DNN. As shown, the neural network can be organized into L fully-connected 'layers' ($i = 1, ..., L$) with n_i nodes (or *artificial neurons*) per layer that function together to make a prediction. The connections between layers $i-1$ and i are represented by numerical weights, stored in matrix W_i of size $n_i \times n_{i-1}$, and vector b_i of length n_i. Thus, if the input values for layer i, given by the values at the n_{i-1} nodes of layer $i-1$, are represented as a vector a_{i-1} of size n_{i-1}, the output of layer i will be a vector of size n_i, given by the matrix-vector product $W_i a_{i-1} + b_i$. As training will be done in parallel for a batch of nb vectors, the input matrices A_{i-1} are of size $n_{i-1} \times nb$ and the outputs are given by the matrix-matrix products $Z_i = W_i A_{i-1} + b_i$, where "+" adds b_i to each of the nb columns of the resulting matrix. The *Forward propagation* process, given by steps $0, ..., L$, represents a non-linear hypothesis/prediction function $H_{W,b}(X) \equiv A_L$ for given inputs of X and fixed weights W, b. The weights must be modified so that the predictions $H_{W,b}(X)$ become close to given/known outcomes stored in Y. This is known as a *classification* problem and is a case of so called *supervised learning*. The modification of the weights is defined as a minimization problem on a cost function J, e.g.,

$$\min_{W,b} J(W,b), \text{ where } J(W,b) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log H_{W,b}(x_i) + (1-y_i)\log(1 - H_{W,b}(x_i)).$$

This is solved by a batch SGD that uses a batch of nb training examples at a time. The derivatives of J with respect to the weights (W and b) are derived over the layers using the chain rule for differentiating compositions of functions. They are computed then by the *backward propagation* steps $L+1$, ..., $2L$, and used to modify their respective weights W_i, b_i during the iterative training process for each layer i as:

$$W_i = W_i - \lambda dW_i, \quad b_i = b_i - \lambda db_i,$$

where λ is a hyperparameter referred to as *learning rate*. The $\sigma_1, ..., \sigma_L$ functions are the activation functions for the different layers of the network, and σ' are their derivatives. The ".*" notation is for point-wise multiplication. The case of $nb = 1$ is the standard (synchronous) SGD. As illustrated, the main kernels are GEMMs, other BLAS, simple auxiliary LA kernels, and various activation functions, which are accelerated using the MAGMA backend.

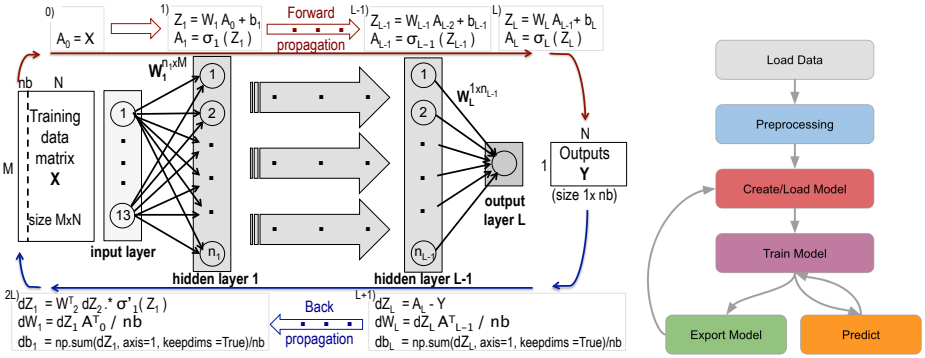

Fig. 2. Typical DNN computational kernels (Left) and MagmaDNN workflow (Right).

MagmaDNN is capable of many different workflows, however it is designed towards the linear one depicted in Figure 2, Right. Each level of the workflow is supported by some functionality in MagmaDNN, however, currently the *Train Model* step is the focus of development.

3 Hyperparameter optimization

MagmaDNN uses a Random and/or Exhaustive Grid Search technique to optimize hyperparameters. The routine is modular and able to add new dimensions to the search space. In grid search, a parameter server sends a parameter set to each node, where the model is trained according to its received parameters. The parameter server, or master, in turn receives the training time, accuracy, and loss associate with each parameter set. Using some objective function, typically a combination of training duration and accuracy, the optimization routine gives the optimal training parameters. Grid search can be run to exhaustively search

a range of parameters (with a given step size). MagmaDNN provides the full search capability as one option. However, this is often too large to be feasible. For these reasons grid search can be ran using random sampling until some accuracy threshold is met. We use the openDIEL framework [17], described next, to run the hyperparameter search in parallel.

3.1 openDIEL Framework Design Overview

The openDIEL system consists of a library that contains all of the function needed to manage modules. Typically, a main driver file is created which contains all of the needed function calls to set up the main MPI communicator that the IEL library uses, set up necessary modules for tuple space communication, and calling the user defined modules.

The main way that users interact with the system is through a configuration file. There are two major components of the openDIEL system: the executive library, and the communication library (see Figure 3).

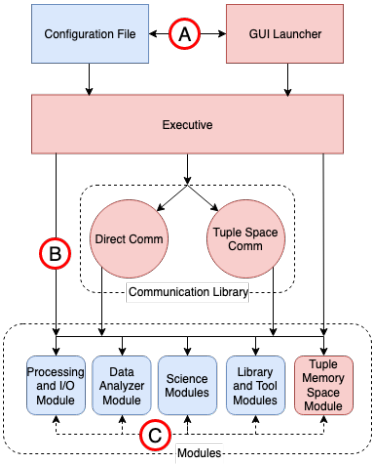

Fig. 3. openDIEL architecture: (A) GUI launcher creates a configuration file for the workflow, and `executive` will read this file to set up workflows; (B) After initial configuration, `executive` starts all modules; (C) The modules have access to the communication library, and directly communicate or utilize tuple space communication.

Configuration File Information about how modules communicate and rely on one another is contained in a configuration file. The configuration file defines what resources the modules requires, such as the number of cores, and number of GPUs required by the module. After defining the modules themselves, a section of the file subsequently defines the manner in which groups of modules depend on one another, how many iterations need to run, amongst other characteristics.

Communication Library The communication library is essentially a wrapper around various MPI calls, and is responsible for managing both tuple space and direct communication between modules. This is done by creating a main MPI_COMM_WORLD communicator in which all of the modules run, and then subdividing this main communicator at the level of single modules. If there are multiple concurrent copies of the same module running at the same time, the module sub-communicator is further subdivided between the copies [17].

Two different methods of communication are provided by an API: tuple space communication, and direct module-to-module communication. With tuple space communication, a tuple server module is used that allows modules to concurrently send data to and receive data from a shared associative array. Modules can use this form of communication to send and receive data from the tuple space respectively. Each module that puts data into the tuple space can issue a non-blocking (IEL_tput) function call, and provide a tag for the data placed in the tuple space. The receiving module can use a blocking (IEL_tget) function call to retrieve the data with the specified tag.

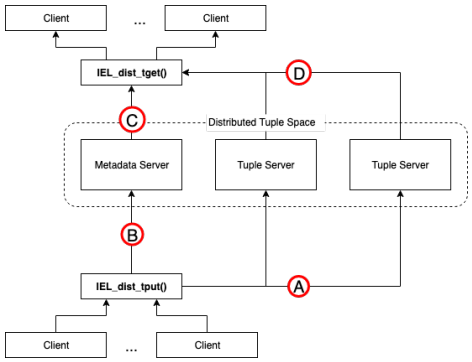

Fig. 4. Distributed tuple server model: (A) Client data is distributed across an array of tuple servers; (B) Metadata for the distributed data is stored in a metadata server; (C) When data is to be retrieved, first the metadata is retrieved, and (D) the data itself is retrieved from the distributed array of tuple servers. The received data is then reconstructed and returned to the requesting client.

The tuple space can also be distributed across a number of different modules, essentially providing a way to store and retrieve data in a distributed manner. The functions IEL_dist_tget and IEL_dist_tget utilize this multiple tuple server model. The IEL_dist_tget function will take a pointer to data and a string to tag the data, and distribute it amongst an array of tuple servers. Information about the distribution of the data is stored on a meta-data server. The IEL_dist_tget function will retrieve the data by querying the meta-data server, which returns the locations of the servers holding the data, the data servers are queried, and the stored data is reconstructed.

Executive Library The executive library is the other major part of the library responsible for starting job and managing dependencies. When a job starts, the executive will read in a workflow configuration file, and then based on this file, the executive will create a dependency graph of the specified workflow, and then start modules based on the graph. Typically, a module is included in openDIEL by linking a library against a driver file, and function pointers are provided to the executive so that they can be called with the appropriate arguments [17]. Executables can also be run by calling fork() and exec() in an MPI process, but limits the ability of the module to use the inter-modular communication provided by openDIEL.

Typical Usage Typically, all of the needed functions are called in a main driver file. This driver file will call MPI_init, and then it will call openDIEL member function IELAddModule, which will take a pointer to the function in the linked library for the module. This will be used later to start the module in the workflow. For modules that are executables, a model that calls fork() and exec() on the proper arguments is started for each serial module. After this setup, the main IELExecutive() member function is called. This function will split up the MPI_COMM_WORLD communicator into the appropriate subcommunicators, resolve dependencies from the configuration file, and then start modules.

3.2 Grid Engine

One of the goals of the framework is to not only provide facilities for hyperparameter optimization and search, but also to allow for users to readily use existing libraries to perform these tasks. One such implementation is a grid search engine that uses the openDIEL tuple space communication to distribute parameters to worker processes in an exhaustive search of a specified parameter space, collect the results, and report the best parameters found.

The module consists of a master process that chooses hyperparameters, and a set of processes that receive the hyperparameters. The master process first selects a set of parameters, distributes them to the workers via the tuple space, and waits for the group to finish. The group of worker processes receive the parameters, train, and report their results to the trainer via tuple space communication. These results are then gathered by the master process, and then the next group of processes is started on the next batch of parameters.

4 Performance results

4.1 MNIST Test

Since its introduction by Lecun et. al. [11] the MNIST data set has become a standard for learning and training neural networks. The data set consists of 60000 images of handwritten digits that are 28x28 pixels in size.

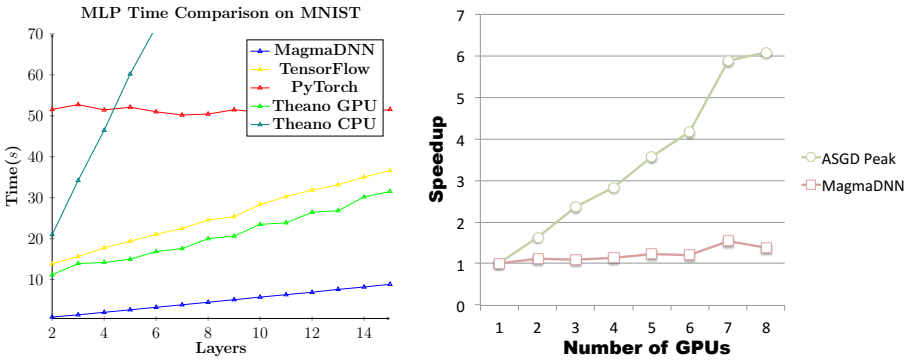

Fig. 5. Left: Time comparisons of MagmaDNN training to other popular frameworks on a single GPU. **Right**: Scalability and MagmaDNN SGD speedup and peak asynchronous SGD (ASGD) speedup vs. TensorFlow SGD training.

As a baseline test MagmaDNN was compared to Tensorflow, PyTorch, and Theano (see Figure 5, Left) on a dense network using the MNIST data set. The layers were increased with each test to show how each framework scaled in training time. A learning rate of $\eta = 0.05$, a weight decay of $\alpha = 0.001$ and activation function `sigmoid` were used in training the network. Data was loaded into the network with a batch size of 100 samples and ran for 5 epochs. The tests were conducted on an Intel Xeon X560 processor alongside an Nvidia 1050 Ti GPU card. Theano was also run on CPU only to give reference to the speedups gained by using GPUs.

From the above test MagmaDNN was the fastest at each data point. It ran approximately 6.8× faster than TensorFlow, and 17.8× faster than Theano-CPU. The performance results shown are averaged over five runs.

4.2 Scaling

Despite being the fastest MagmaDNN scaled the second fastest in terms of training time. PyTorch, being the fastest scaling, performed poorly on the small data set, but did not gain much training time as the network size increased.

Framework	**Δ time / Δ layer**
MagmaDNN	0.6197
TensorFlow	1.7524
Theano (GPU)	1.5271
Theano (CPU)	12.5071
PyTorch	-0.08

Table 1. Change in Training Time with Number of Layers

Due to the computational size of deep networks much effort has been put into distribution strategies. Parallelization introduces speed increases, but can also hurt convergence.

As evinced by Figure 5, GPUs provide a significant performance boost in training deep networks. Thus making full use of the GPU is vital for a new deep learning framework.

Even with the advent of GPUs in training, they are still insufficient for training larger networks (such as ResNet-50 or DenseNet) in a reasonable amount of time. These large networks typically train on vast data sets causing memory transferring to bottle neck the training. One solution to this problem is to use minibatches. Batches reduce the total number `cudaMemcpy` calls. Using batches also introduces additional optimization complexity by creating a new hyperparameter *batch size*.

In addition to adding new hyperparameters, using large batch sizes can also hinder the convergence of the network. In order to combat the poor convergence, tricks such as growing batch sizes [12], warm-up [8], or layer-wise adaptive rate scaling (LARS) [18] must be used. In practice these tricks are often successful, but only raise the batch size barrier [3]. Using these techniques and various others You et al. were able to train AlexNet in 11 minutes on the ImageNet-1k data set [19].

All of the above techniques are typical to modern deep learning approaches, however, they do not address multi-node training. DNN parallelization can also be implemented to accelerate training, while retaining convergence.

The most common of these techniques is *Data-Parallelism* (see Figure 6). Older implementations of data parallelism use a master-worker model (see Figure 6, Left) for averaging weights. In this model weights are sent from a master node to N worker nodes. Let w^j be the weights of the j-th worker node. Each node computes the gradient ∇w^j and sends it back to the master node. Once the master node has received the gradients from each worker it calculates $\overline{w} \leftarrow \overline{w} - \eta/N \sum_{j=1}^{N} \nabla w^j$, the average weight, and broadcasts $\overline{w}$ back to each worker. Modern implementations, as well as MagmaDNN's implementation, remove the Master node and average the gradients using `AllReduce`. Any CUDA-aware MPI implementation can perform this operation, however, Nvidia's NCCL has in general a much faster `ncclAllReduce` between GPU nodes. Data parallelism is a typical method used in scaling deep learning and it has shown promising results [9][4].

While providing significant speed ups, data parallelism also provides some drawbacks. In the `Allreduce` approach nodes that finish training early sit idle while waiting on others. This creates "lulls" where more work could be done. To address this issue some parallel trainers utilize *Model-Parallelism*. Here models are partitioned across nodes and trained in parallel. This approach can quickly fill up device memory, thus restricting itself to smaller models and/or batch sizes.

Layer-Parallelism aims to solve the idle processor issue by pipelining network layers, computing layers in parallel as soon as possible. Layer parallelism offers

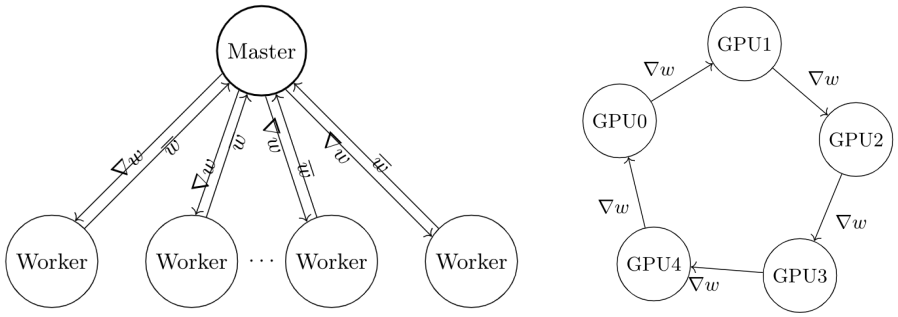

Fig. 6. Master-Worker Reduce (Left) and Ring AllReduce (Right).

some performance benefits and is used in practice [10][1], but creates irregular transfer rates between processors [3].

As each distribution strategy offers unique solutions and drawbacks the best strategy is *Hybrid-Parallelism*, which combines each of the previous in some custom manner to exploit the parallel nature of a specific model. However, this makes hybrid parallelism model specific and non generalizable.

MagmaDNN makes use of MAGMA to exploit best fine-grained parallel practices. CUDA and CuDNN are additionally utilized to exploit the highly parallel GPU architectures. Other fine-grained acceleration is currently not within the projects scope.

Techniques such as data parallelism and distributed training with CUDA-aware MPI are included in MagmaDNN to employ course-grained parallelism.

MPI is utilized to distribute networks across nodes in MagmaDNN using `Allreduce` to implement data parallelism. Despite MPI not being fault-tolerant training is typically not hindered due to the large number of samples trained on and the resilience of deep networks.

Figure 5, Right shows the speedup of MagmaDNN's SGD training vs. TensorFlow train- ing on a system with up to 8 V100 GPUs. In this case MagmaDNN outperforms TensorFlow by about 50%. Shown also is a comparison to a peak performance of an asynchronous SGD. The large speedups illustrate the high potential that asynchronous methods have for accelerating the computation.

4.3 Hyperparameter optimizations

We tested the hyperparameter optimization framework on a number of applications. Most notably, we used it as a proof of concept in the design, evaluation, and optimization of DNN architectures of increasing depth. For example, when applied to heart disease diagnosis [14], the hyperparameter optimization led to the discovery of a novel five layer DNN architecture that yields best prediction accuracy (using the publicly available Cleveland data set of medical information and a predefined search space), e.g., yielding 99% accuracy and 0.98 Matthews

correlation coefficient (MCC), significantly outperforming currently published research in the area [14].

5 Conclusions and future directions

As the availability of exaflop computing capabilities approaches, deep learning continues to be far from utilizing the entirety of available computing power. Thus, it is crucial to continue and pursue distribution strategies and techniques for training deep networks on clusters and supercomputers.

MagmaDNN, due to its HPC MAGMA backend and initial speed, shows potential in becoming a tool for future deep learning applications. Its native C++ interface allows easier integration with existing C and fortran scientific codes. However, MagmaDNN currently lacks the arsenal of features present in other popular frameworks due to its infancy.

MagmaDNN aims to continue to add the necessary features for a full deep learning suite, while maintaining a fast scalable interface. Future development will focus on performance enhancements in distributed training, while providing a modular framework that allows for customization and tuning. Interfaces and ease of use are also very important, and to be able to compete with other frameworks, we are considering adding Python APIs to MagmaDNN.

6 Availability

MagmaDNN is currently developed and supported by the Innovative Computing Laboratory (ICL) and Joint Institute for Computer Science (JICS) at the University of Tennessee, Knoxville and Oak Ridge National Laboratory. Source code, documentation, tutorials, and licensing can all be found on the project's homepage[3].

Acknowledgments

This work was conducted at the Joint Institute for Computational Sciences (JICS) and the Innovative Computing Laboratory (ICL). This work is sponsored by the National Science Foundation (NSF), through NSF REU Award #1659502, with additional Support from the University of Tennessee, Knoxville (UTK), the National Institute for Computational Sciences (NICS), and NSF Awards #1740250 and #1709069. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant #ACI-1548562. Computational Resources are available through a XSEDE education allocation awards TG-ASC170031 and TG-ASC190013. In addition, the computing work was also performed on technical workstations donated by the BP High Performance Computing Team, as well as on GPUs donated by NVIDIA.

[3] https://bitbucket.org/icl/magmadnn/

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I.J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D.G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P.A., Vanhoucke, V., Vasudevan, V., Viégas, F.B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. CoRR **abs/1603.04467** (2016), http://arxiv.org/abs/1603.04467
2. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.J.: Performance, design, and autotuning of batched GEMM for gpus. In: High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings. pp. 21–38 (2016). https://doi.org/10.1007/978-3-319-41321-1_2, https://doi.org/10.1007/978-3-319-41321-1_2
3. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. CoRR **abs/1802.09941** (2018), http://arxiv.org/abs/1802.09941
4. Chen, J., Monga, R., Bengio, S., Józefowicz, R.: Revisiting distributed synchronous SGD. CoRR **abs/1604.00981** (2016), http://arxiv.org/abs/1604.00981
5. Chen, S., Gessinger, A., Tomov, S.: Design and Acceleration of Convolutional Neural Networks on Modern Architectures. Tech. rep., Joint Institute for Computational Sciences (JICS), UTK (2018), 2018 Summer Research Experiences for Undergraduate (REU), Knoxville, TN, 2018.
6. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. CoRR **abs/1410.0759** (2014), http://arxiv.org/abs/1410.0759
7. Gates, M., Tomov, S., Dongarra, J.: Accelerating the svd two stage bidiagonal reduction and divide and conquer using gpus. Parallel Computing **74**, 3 – 18 (2018). https://doi.org/https://doi.org/10.1016/j.parco.2017.10.004, http://www.sciencedirect.com/science/article/pii/S0167819117301758, parallel Matrix Algorithms and Applications (PMAA'16)
8. Goyal, P., Dollár, P., Girshick, R.B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K.: Accurate, large minibatch SGD: training imagenet in 1 hour. CoRR **abs/1706.02677** (2017), http://arxiv.org/abs/1706.02677
9. Iandola, F.N., Ashraf, K., Moskewicz, M.W., Keutzer, K.: Firecaffe: near-linear acceleration of deep neural network training on compute clusters. CoRR **abs/1511.00175** (2015), http://arxiv.org/abs/1511.00175
10. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R.B., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. CoRR **abs/1408.5093** (2014), http://arxiv.org/abs/1408.5093
11. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (Nov 1998). https://doi.org/10.1109/5.726791
12. Smith, S.L., Kindermans, P., Le, Q.V.: Don't decay the learning rate, increase the batch size. CoRR **abs/1711.00489** (2017), http://arxiv.org/abs/1711.00489
13. Sorna, A., Cheng, X., D'Azevedo, E., Wong, K., Tomov, S.: Optimizing the fast fourier transform using mixed precision on tensor core hardware. In: 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW). pp. 3–7 (Dec 2018). https://doi.org/10.1109/HiPCW.2018.8634417

14. Tomov, N., Tomov, S.: On deep neural networks for detecting heart disease. CoRR **abs/1808.07168** (2018), http://arxiv.org/abs/1808.07168
15. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid gpu accelerated manycore systems. Parallel Computing **36**(5), 232 – 240 (2010). https://doi.org/https://doi.org/10.1016/j.parco.2009.12.005, http://www.sciencedirect.com/science/article/pii/S0167819109001276, parallel Matrix Algorithms and Applications
16. Tomov, S., Haidar, A., Ayala, A., Schultz, D., Dongarra, J.: Design and Implementation for FFT-ECP on Distributed Accelerated Systems. ECP WBS 2.3.3.09 Milestone Report FFT-ECP ST-MS-10-1410, Innovative Computing Laboratory, University of Tennessee (April 2019), revision 04-2019
17. Wong, K., Brown, L., Coan, J., White, D.: Distributive interoperable executive library (diel) for systems of multiphysics simulation. In: 2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 49–55. IEEE (2014)
18. You, Y., Gitman, I., Ginsburg, B.: Scaling SGD batch size to 32k for imagenet training. CoRR **abs/1708.03888** (2017), http://arxiv.org/abs/1708.03888
19. You, Y., Zhang, Z., Hsieh, C., Demmel, J.: 100-epoch imagenet training with alexnet in 24 minutes. CoRR **abs/1709.05011** (2017), http://arxiv.org/abs/1709.05011