2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-9747-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPSW55747.2022.00177

Adaptive Optimization for Sparse Data on Heterogeneous GPUs

Yujing Ma, Florin Rusu
University of California Merced
{yma33,frusu}@ucmerced.edu

Kesheng Wu, Alexander Sim
Lawrence Berkeley National Lab
{kwu,asim}@lbl.gov

***Abstract*—Motivated by extreme multi-label classification applications, we consider training deep learning models over sparse data in multi-GPU servers. The variance in the number of non-zero features across training batches and the intrinsic GPU heterogeneity combine to limit accuracy and increase the time to convergence. We address these challenges with Adaptive SGD, an adaptive elastic model averaging stochastic gradient descent algorithm for heterogeneous multi-GPUs that is characterized by dynamic scheduling, adaptive batch size scaling, and normalized model merging. Instead of statically partitioning batches to GPUs, batches are routed based on the relative processing speed. Batch size scaling assigns larger batches to the faster GPUs and smaller batches to the slower ones, with the goal to arrive at a steady state in which all the GPUs perform the same number of model updates. Normalized model merging computes optimal weights for every GPU based on the assigned batches such that the combined model achieves better accuracy. We show experimentally that Adaptive SGD outperforms four state-of-the-art solutions in time-to-accuracy and is scalable with the number of GPUs.**

I. INTRODUCTION

Deep learning models have been shown to achieve high accuracy for many classification problems across diverse application domains, including speech recognition [10], finance [8], and renewable energy [19]. However, this is heavily dependent on the amount of training data and the size of the model. As these two values increase, building accurate deep learning models becomes time-consuming even on specialized hardware accelerators such as GPUs [9] and TPUs [24] because of their reduced memory—which requires many slow data transfers with the CPU [4]. Thus, in order to scale training, parallel processing across multiple GPUs [6], [13] becomes a necessity. This approach is facilitated by the preponderance of multi-GPU computing architectures both on supercomputers [30] and in the cloud [28].

Extreme Multi-label Classification (XML). We consider extreme multi-label classification as a motivating application for our work. The objective in XML is to tag a data point with the most relevant subset of labels from an extremely large set that can contain up to millions of possible labels. The Extreme Classification Repository [2] contains an exhaustive collection of real datasets, code, and experimental results for XML algorithms. In a typical dataset (Table I), the feature vector, as well as the associated labels, are highly sparse, thus processing requires sparse linear algebra operations. This is quite different from the image classification benchmarks – which are based on dense linear algebra – used almost exclusively to evaluate the existing multi-GPU training algorithms [9], [25], [13]. AttentionXML [23], DeepXML [7], and LightXML [12] are some of the most recent deep learning XML algorithms that achieve the highest accuracy. Although these algorithms use complex learning architectures, such as the transformer model, tree-based models, and generative cooperative networks, they achieve top-1 accuracy below 50% even after training for hours on multi-GPU servers.

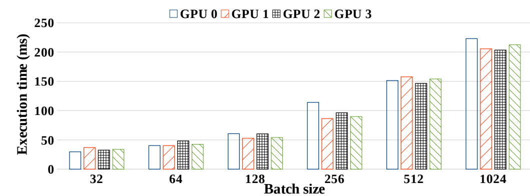


Fig. 1: Multi-GPU heterogeneity on training a deep learning model with an identical batch of sparse data.

Heterogeneity in Multi-GPU Architectures. We argue that an important reason for the high XML training time is the uniform handling of multiple GPUs. There are two major sources of heterogeneity in a multi-GPU architecture for training on sparse data. The first source is due to the differences among identical GPUs. The clock rate and memory latency display oscillations on GPUs with the same model from the same vendor. This effect is amplified when multiple GPUs are integrated on the same server, in which case the execution time varies within observable ranges. For example, given the same training batch, the gap between the fastest and slowest GPU is as large as 32% when performing an epoch of the training algorithm on a server with 4 NVIDIA V100 GPUs (Figure 1). The second source of

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00177

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

heterogeneity is specific to sparse data. The number of non-zero features varies significantly among the training samples. When we partition training data into batches, it is almost impossible to guarantee that every batch contains the same number of non-zeros. Since sparse linear algebra operations are sensitive to the cardinality of their input, the effect is variation in processing across batches. The combination of these two factors results in significant differences among GPUs, which impacts negatively the time-to-accuracy of the training algorithm.

Problem. We investigate multi-GPU training algorithms for deep learning. We consider two dimensions of this problem that have not been adequately addressed before. We target sparse training data with sparse labels having high dimensionality as found in XML classification tasks. The second dimension is the heterogeneity of the environment consisting of GPUs that exhibit significant variance in executing identical tasks. When data sparsity and GPU heterogeneity are combined, the delay among GPUs at the synchronization barriers becomes a severe bottleneck that can hinder time-to-accuracy dramatically. Solutions focused exclusively on determining the optimal synchronization frequency are challenging since they depend both on the data and the relative GPU performance. Instead we target adaptive algorithms that monitor the execution continuously and configure the workload for every GPU according to its relative performance. This introduces variation across the local models trained by different GPUs, which requires the design of an appropriate model merging scheme.

Contributions. We introduce Adaptive SGD – an adaptive elastic model averaging stochastic gradient descent algorithm for heterogeneous multi-GPUs – characterized by dynamic scheduling, adaptive batch size scaling, and normalized model merging:

- Instead of statically assigning batches to GPUs, in dynamic scheduling batches are allocated based on the relative GPU processing speed. This process is controlled by fixing the number of training samples processed between two model merging stages.
- Since execution-driven allocation can lead to a different number of model updates across GPUs – which is problematic for accuracy – batch size scaling assigns larger batches to the faster GPUs and smaller batches to the slower ones, with the goal to arrive at a steady state in which all the GPUs perform the same number of model updates. The batch sizes are continuously updated following a linear function that quantifies the deviation from the expected number of updates, while guaranteeing a minimum degree of GPU utilization and imposing strict bounds on model replica staleness.
- Normalized model merging computes optimal weights for every GPU based on the assigned batches. The underlying principle is to prioritize the replicas updated more frequently and – secondarily – with gradients derived from larger batch sizes. In order to increase the importance of the relevant model replicas, perturbation is added to the normalized weights when the replicas are well-regularized.

We implement Adaptive SGD in a new framework for sparse deep learning on multiple GPUs and compare its performance against four alternative methods. Due to the careful handling of heterogeneity – which allows a more thorough exploration of the optimization space – Adaptive SGD outperforms all the competitors in time-to-accuracy. In fact, Adaptive SGD always achieves the highest accuracy among all the algorithms. Moreover, as we increase the number of GPUs, Adaptive SGD exhibits both faster time-to-accuracy and less epochs to achieve an accuracy target. This confirms its superior scalability.

II. PRELIMINARIES

Stochastic Gradient Descent (SGD). SGD is the most common method to train deep learning models [3]. At high-level, SGD iteratively computes the gradient of the loss function – which is a measure of the error between the predicted and the true labels – over the training dataset and moves the model in the opposite direction of the gradient—which results in a decrease of the error. In practice, the gradient is only estimated from a batch of training samples. SGD requires three passes over the DNN. In the forward pass, the predicted labels are computed based on the current model. The backward pass implements the chain rule of calculus for computing the gradient of a composite function starting from the predicted labels. The third pass – which is also forward – updates the model with the computed gradient using the learning rate hyperparameter. SGD can be stopped either after a fixed number of iterations, i.e., *epochs*, or when there is no significant drop in the error.

The performance of a deep learning model is assessed by measuring its error/accuracy in predicting the labels of a test dataset that is not used in training. This measure is known as test accuracy. The goal of training is to reduce the time to reach a target level of test accuracy, i.e., time-to-accuracy. Two factors determine the time-to-accuracy. The first is the number of epochs required by SGD, known as statistical efficiency, while the second factor is the execution time of an epoch—known as hardware efficiency. These two factors are themselves dependent

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

on the hyperparameters – batch size and learning rate – of the SGD algorithm.

The standard approach to parallelize SGD is to partition the samples across the available GPUs and allocate a separate model replica to every GPU. Model replica transfer and synchronization become the dominant challenges in this setting. Depending on how they are addressed, two parallel SGD strategies are possible.

Gradient Aggregation. Gradient aggregation – or synchronous SGD [5] – requires all the GPUs to have an identical replica of the model at every epoch. This guarantees that the same model is used to compute the gradient by every GPU and local gradients can be aggregated following a sound formula. In an epoch, every GPU computes a partial gradient from a batch of training samples having the same size across all the GPUs and the identical model replica. The partial gradients are aggregated – summed or averaged – by coordinating all the GPUs. The aggregated gradient is shared among all the GPUs and applied to update every local model replica. Once this is done, the next epoch can begin. The synchronization imposed by gradient aggregation at every epoch is the main limitation of synchronous SGD—known as the straggler problem [13]. Asynchronous SGD [17], [21], [18], [15], [1] transforms gradient aggregation into a completely asynchronous process in which a GPU transitions to the next epoch immediately after its partial gradient is added to the aggregated gradient. The value accumulated thus far is used to update its local model replica. However, if performed over a large number of epochs, asynchronous SGD can result in poor convergence.

Elastic Model Averaging. The elastic model averaging strategy [26] – or K-step averaging [27] – is a trade-off between synchronous and asynchronous SGD. It alleviates the burden in synchronous SGD by performing synchronization only after a predefined number of epochs—or batches. The model divergence issue from asynchronous SGD is addressed by averaging model replicas instead of gradients. By controlling the synchronization frequency, elastic model averaging generalizes gradient aggregation. It becomes synchronous SGD when aggregation is performed after every epoch—notice that gradient and model averaging are equivalent in this case. When the frequency is set to infinity and the local gradient/model replicas are references to a shared global gradient/model, elastic model averaging morphs into asynchronous SGD. While the synchronization cost is amortized, this does not eliminate the straggler problem since the delay among GPUs is cumulated across epochs. The delay depends both on the discrepancy between GPUs, as well as on data sparsity, and can vary from one epoch to another. In general, the reduction over gradient aggregation is inversely proportional to the synchronization frequency—the higher the frequency, the lower the reduction.

III. ADAPTIVE SGD

Elastic model averaging imposes a strict requirement that every GPU has to process the same number of batches with the same size between two model averaging stages. This guarantees that every model replica is updated the same number of times with gradients approximated from samples with the same cardinality. Thus, averaging is straightforward. In a heterogeneous multi-GPU environment, this requirement exacerbates the straggler problem by increasing the waiting time of the faster GPUs and reducing their utilization. Ultimately, this increases the time-to-accuracy.

We address the limitation of elastic model averaging with continuous monitoring of the GPU execution, dynamic scheduling, and adaptive batch size scaling. Since monitoring is performed only at the synchronization points, its overhead is negligible because it overlaps with model transfer and merging. Instead of statically assigning batches to GPUs for every model averaging stage, in dynamic scheduling batches are dispatched one-by-one based on GPU availability. Whenever a GPU finishes an epoch, it is assigned the next batch. Similar to elastic model averaging, this process is controlled by fixing the number of training samples processed between model merging stages—we call these samples a mega-batch. As depicted in Figure 2, dynamic scheduling can lead to a different number of model updates across GPUs—3 updates for GPU 1 and 2 updates for GPU 2. This creates an impediment for model averaging because the replicas are not on the same time horizon. In our example, the replica on GPU 1 is one step ahead of the replica on GPU 2. In order to mitigate the stale replica problem, we design an adaptive batch size scaling mechanism that is performed during model merging. The goal of batch size scaling is to assign larger batches to the faster GPUs and smaller batches to the slower ones, such that they perform the same number of model updates—as shown for the second mega-batch in Figure 2. While this brings the replicas on the same time horizon, it also requires significant changes to the SGD algorithm. First, the batch size corresponding to every GPU has to be adequately determined. The dynamic modification of the batch size also requires an update of the corresponding learning rate. Thus, both of these hyperparameters

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

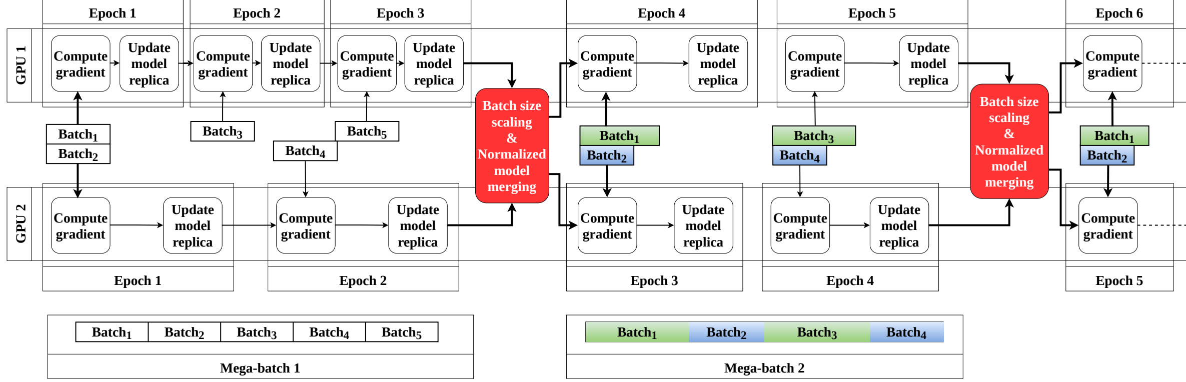


Fig. 2: Adaptive elastic SGD on heterogeneous multi-GPUs.

become instances associated with every GPU, rather than being global for the SGD algorithm. This leads to the second change. Model merging has to carefully consider the differences across replicas, including the hyperparameters – batch size and learning rate – as well as the magnitude of the model parameters. We discuss the technical details of the resulting Adaptive SGD algorithm in the following sections.

Algorithm 1 Batch Size Scaling

Input: current batch size b_i and learning rate lr_i
number of model replica updates u_i for GPU i
minimum b_{min} and maximum b_{max} batch size
batch size scaling parameter β
Output: updated batch size b_i and learning rate lr_i
1: Average model updates: $\tilde{\mu} \leftarrow \left(\sum_i u_i\right) / |GPU|$
▷ **Scale batch size and learning rate**
2: **for all** GPU_i **do**
3: **if** $u_i > \tilde{\mu}$ **and** $b_i + \beta \cdot (u_i - \tilde{\mu}) \leq b_{max}$ **then**
4: $lr_i \leftarrow lr_i \cdot (b_i + \beta \cdot (u_i - \tilde{\mu}))/b_i$
5: $b_i \leftarrow b_i + \beta \cdot (u_i - \tilde{\mu})$
6: **else if** $u_i < \tilde{\mu}$ **and** $b_i - \beta \cdot (\tilde{\mu} - u_i) \geq b_{min}$ **then**
7: $lr_i \leftarrow lr_i \cdot (b_i - \beta \cdot (\tilde{\mu} - u_i))/b_i$
8: $b_i \leftarrow b_i - \beta \cdot (\tilde{\mu} - u_i)$
9: **end for**

A. Batch Size Scaling

The batch size scaling procedure is presented in Algorithm 1. The goal is to arrive at a steady state in which all the GPUs perform the same number of replica updates. By default, the algorithm is executed after every mega-batch. However, if stability is achieved or the system enters an oscillatory state, the frequency at which scaling is performed can be increased. The algorithm targets to compute an updated batch size b_i and learning rate lr_i for every GPU i based on their corresponding number of model updates u_i. Since the update to the learning rate is completely determined by the linear scaling rule introduced in [9], the main challenge is how to update the batch size. After experimenting with several functions, we settle for linear update with a parameter β determined empirically. The batch size b_i is increased/decreased by a factor proportional to the deviation of the number of updates u_i from the average number of updates $\tilde{\mu}$ across all GPUs.

In order to guarantee a minimum degree of GPU utilization, the minimum batch size is bounded by a threshold b_{min}. A symmetric bound b_{max} controls the maximum size of a batch that fits in the GPU memory. b_{min} and b_{max} are derived from the mega-batch size such that the discrepancy between the number of model updates performed across GPUs is restricted. Otherwise, some GPUs dominate the SGD algorithm, rendering the others ineffective—they can be removed without impacting the SGD time-to-accuracy. Moreover, b_{min} and b_{max} allow for analytical characterization of batch size scaling. Assuming an equal number of model updates across GPUs, the convergence behavior of SGD with batch size scaling is within the range of elastic model averaging with a batch size between b_{min} and b_{max}. When the number of updates varies, these thresholds impose bounds on replica staleness, allowing the application of convergence results from stale synchronous SGD [11], [14].

B. Normalized Model Merging

While batch size scaling aims to arrive at the same number of model replica updates across all the GPUs, this cannot be achieved instantaneously—or at all. Thus,

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Normalized Model Merging

Input: current global model $\overline{w}$, previous global model $\overline{w_p}$, model replica $\overline{w}_i$ for every GPU i
batch size b_i and learning rate lr_i
number of model replica updates u_i
perturbation threshold $pert_{thr}$ and factor δ
momentum hyperparameter γ
Output: updated global model $\overline{w}$
$\triangleright$ **Compute the normalization weights**
1: Let α_i be the weight corresponding GPU i
2: **if** $u_i = u_j, \forall i, j \in GPU$ **then** $\alpha_i \leftarrow b_i / \left(\sum_i b_i\right)$
3: **else** $\alpha_i \leftarrow u_i / \left(\sum_i u_i\right)$
$\triangleright$ **Add perturbation to min/max weights**
4: **if** $||\overline{w}_i||_2 / |\overline{w}| < pert_{thr}, \forall i \in GPU$ **then**
5: $\quad r \leftarrow \text{argmax}_i \{u_i\}$, $s \leftarrow \text{argmin}_i \{u_i\}$
6: $\quad \alpha_r \leftarrow (1+\delta) \cdot \alpha_r$, $\alpha_s \leftarrow (1-\delta) \cdot \alpha_s$
7: **end if**
$\triangleright$ **Compute and update the global model**
8: $\overline{w'} \leftarrow \sum_i \alpha_i \cdot \overline{w_i} + \gamma \cdot (\overline{w} - \overline{w_p})$
9: $\overline{w_p} \leftarrow \overline{w}$, $\overline{w} \leftarrow \overline{w'}$

model merging has to consider both cases with and without identical number of updates when deriving the global model from the local replicas. Moreover, merging has to account for the difference in batch size. Algorithm 2 is a general model merging procedure that handles these cases methodically. The underlying principle is to prioritize the replicas updated more frequently and – secondarily – with gradients derived from larger batch sizes. This leads to weighted average model merging, where the weights are normalized either based on the batch size – when the number of model updates are identical – or the number of updates itself. In the first case, larger batch sizes generate more accurate gradients, thus, more accurate models. For the second case, the choice to normalize exclusively based on the number of model updates requires some discussion. This situation is more likely to appear for the first mega-batches, before batch size scaling guides the Adaptive SGD algorithm to a steady state. Since the model is farther from the global minimum, a wider exploration is beneficial to identify the appropriate direction—and hyperparameters. This is similar to the warmup approach [9]. An alternative for later stages is to normalize based on the product between the number of updates and the batch size.

In order to increase the importance of the most updated replica, Algorithm 2 adds perturbations to the normalized weights. This is done by increasing the weight corresponding to the most updated replica and decreasing the weight for the replica with the fewest updates accordingly. The degree of increase/decrease is controlled by the perturbation factor δ, which is a parameter defined by the user and set by default to 0.1. It is important to notice that this weight modification can result in denormalization—the sum of the weights α is not equal to 1 anymore. In order to restrict the eventual impact of denormalization, we apply weight perturbation only when all the model replicas are well regularized. We quantify model regularization by the L2-norm per model parameter measure, i.e., L2-norm divided by model dimensionality. Since large values of the L2-norm correspond to unregularized models – skewed along one or more dimensions – weight perturbation is applied only when the L2-norm per model parameter is below a threshold $pert_{thr}$ for all the replicas—$pert_{thr}$ is set by default to 0.1 and can be tuned by the user. This ensures that no skewed parameters are amplified by the denormalized weights.

The final step in Algorithm 2 is to update the global model. This process consists in computing the weighted average of the local replicas using the normalized – and perturbed – weights. Rather than directly assigning this value to the global model, we follow the SGD momentum update rule [20], in which the current and previous global model are combined with the merged replicas. Momentum preserves a stable convergence trajectory while avoiding reactive oscillations. The scale of the past models is controlled by the momentum hyperparameter γ—set to 0.9 according to the literature [20].

IV. IMPLEMENTATION

The Adaptive SGD algorithm for heterogeneous multi-GPUs is implemented in the HeteroGPU framework for sparse deep learning. In this section, we present the HeteroGPU architecture and workflow. Then, we discuss a series of optimizations to enhance the performance of the GPU code, including CUDA kernel fusion and multi-stream all-reduce model merging.

Architecture and Workflow. The architecture of HeteroGPU together with the main tasks performed by every component are depicted in Figure 3. HeteroGPU consists of multiple asynchronous GPU managers corresponding to every GPU in the system – four in this example – and a central dynamic scheduler. They are implemented as stand-alone asynchronous threads that communicate through event messages.

The GPU manager is in charge of the resources and execution of its assigned GPU. It coordinates the data transfers between CPU and GPU, and invokes the CUDA

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

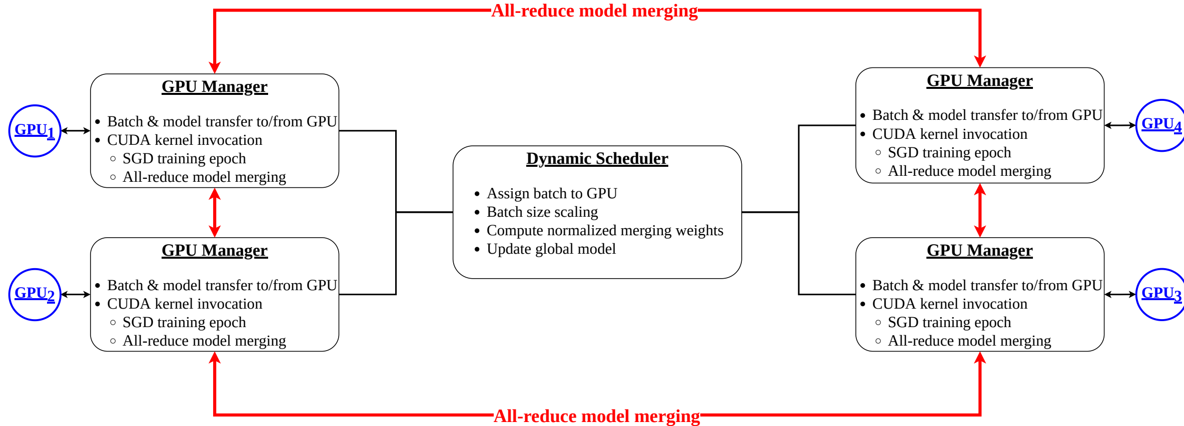


Fig. 3: System implementation architecture and workflow.

kernels. The execution of an epoch of the SGD algorithm requires moving the training batch and the model replica – only at the beginning of a mega-batch – to the GPU and then invoking linear algebra kernels for gradient computation and replica update. Although the GPU manager provides thread isolation for these operations, there is a significant degree of interference in the CUDA environment scheduler – which is shared across GPUs – due to multiple GPU managers launching CUDA kernels simultaneously. The interference manifests by a large kernel startup overhead—which increases with the number of GPUs. HeteroGPU addresses this issue by kernel fusion in event-based asynchronous execution streams. Specifically, small kernels that perform primitive operations, e.g., element-wise linear algebra, are grouped into a large kernel. The GPU manager invokes this kernel in an independent stream and monitors its completion using events that bypass the global CUDA environment. The intermediate output of kernel invocations is kept in the GPU memory in order to reduce data movement. Nonetheless, advanced memory management strategies that work at hidden layer granularity [22] can also be supported. The other major task performed by the GPU manager is model merging at mega-batch boundaries. This is realized with an all-reduce procedure that involves direct data transfers among GPUs.

The most common task of the dynamic scheduler is to assign data batches of different size to the GPU managers. The size of the batches is determined in the batch size scaling procedure—performed at the end of processing a mega-batch. The computation of the normalized model merging weights is also performed at this stage. As shown in Algorithm 1 and 2, these require the number of model replica updates executed by every GPU manager—which are recorded by the scheduler when batches are dispatched. The overhead of these tasks is reduced, making the scheduler a relatively low utilized component. While model merging is performed exclusively by the GPU managers – with the normalized weights computed by the scheduler – model update requires the combination of the merged model with the current and previous global model. This operation can be executed either on every GPU or by the scheduler. We opt for the second alternative because it requires fewer data transfers between CPU and GPU.

All-reduce Model Merging. Model merging is implemented as an all-reduce operation in the HeteroGPU framework. This diminishes the load on the dynamic scheduler, avoiding eventual bottlenecks. In all-reduce model merging, the aggregate – in this case weighted average – is computed through multi-round data partitioning such that all the GPUs end up with the overall aggregated model. While the NCCL [29] multi-GPU communication library provides all-reduce methods, they lack support for multi-streams – which precludes the overlap between model transfer and reduction computation – or are optimized for a multi-server setting—which is not compatible with our single-server scope. Therefore, we implement specialized tree- and ring-based multi-stream all-reduce aggregation functions. The local replica models are split into a fixed number of partitions, which are allocated to a separate GPU processing stream. The optimal number of partitions – and GPU streams – is empirically determined to be equal with the number of GPUs in the system. Every stream performs the all-reduce aggregation starting from a different GPU.

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

dataset	features	classes	training samples	testing samples	avg features per sample	avg classes per sample
Amazon-670k	135,909	670,091	490,449	153,025	76	5
Delicious-200k	782,585	205,443	196,606	100,095	302	75

TABLE I: Experimental datasets for XML classification.

This results in complete overlap between data transfer and computation. While the NCCL tree-based implementation is more efficient on a single stream, the multi-stream ring-based all-reduce function performs model merging at least twice as fast. Thus, this is the method used throughout the experiments.

V. EXPERIMENTAL EVALUATION

We pursue multiple objectives in the experimental evaluation. Due to space limitations, we include only the two most important topics here and direct the reader to the public technical report [16] available online for complete details. First, we assess the overall performance of the proposed Adaptive SGD by comparing it with four well-known baseline algorithms. Second, we check if the specific heterogeneous multi-GPU characteristics of Adaptive SGD have a tangible effect on training. To this end, the questions we ask in the experiments are:

- How does Adaptive SGD compare with Elastic SGD, TensorFlow, CROSSBOW, and SLIDE in terms of time-to-accuracy and statistical efficiency?
- How scalable is Adaptive SGD?
- Do batch size scaling and perturbed model merging get invoked in practice and how often?

A. Setup

Baseline Methods. We include four alternative methods in the experimental evaluation. Three of them are multi-GPU SGD algorithms that do not consider heterogeneity, while the fourth is the SLIDE [31] system, which provides a CPU-optimized SGD algorithm for sparse data. The three multi-GPU alternatives are TensorFlow [32] – which implements gradient aggregation – elastic model averaging, and the CROSSBOW [13] synchronous model averaging. We follow the experimental testbed for XML classification used in the SLIDE system. In addition to the SLIDE implementation, this testbed includes TensorFlow single-GPU code, which we extend to multi-GPUs both with the mirrored and central storage strategy. Since the mirrored strategy proves superior, we include only these TensorFlow results in the paper. We implement the remaining three methods – CROSSBOW, elastic model averaging (Elastic SGD), and the proposed Adaptive SGD – in the HeteroGPU framework, which is a C++ prototype. We do not use the original CROSSBOW implementation because it lacks support for sparse data. The GPU code is written in CUDA 11.2, with primitives from the NVIDIA cuSPARSE library for the sparse linear algebra operations. The training data are stored in the sparse libSVM format.

Datasets and Model. We use two standard datasets for XML classification in our experiments—Amazon-670k and Delicious-200k [2]. Their characteristics are displayed in Table I. Both datasets have a very large number of classes and high-dimensional sparse input features, as shown by the average number of non-zero classes/features per sample. They are used to evaluate the performance of the SLIDE algorithm on a 3-layer Multi-Layer Perceptron (MLP) model having ReLU layer activation, softmax multi-class probability, and cross-entropy loss function. In order to maintain compatibility, we keep the same model configuration in our experiments. The best XML algorithms – which are considerably more complex than the model used in these experiments – achieve top-1 accuracy below 50% on these datasets, while requiring hours to train and GB-size models [2]. Thus, the presented results have to be interpreted in this context.

Methodology. We execute every algorithm for the same amount of time. We measure the top-1 accuracy on the testing dataset after processing every mega-batch. This allows us to compute the time-to-accuracy, statistical efficiency, and hardware efficiency. All the algorithms are initialized with the same model, which gives the same initial loss. The initial values of the model weights are randomly drawn from a normal distribution with standard deviation equal to the number of units in every layer. The initial batch size – set to b_{max} – is chosen such that the GPU memory – and utilization – are maximized. b_{min} is set to a value 8 times smaller than b_{max}, while the batch size scaling parameter β to half of b_{min}. The optimal learning rate for b_{max} is found by griding its range in powers of 10 and selecting the value that achieves the best accuracy across all the algorithms. The learning rates for the other batch sizes are determined based on the linear scaling rule [9]. The same hyperparameters are used for all the algorithms. The time to load the data and evaluate the accuracy are not included in timing measurements.

Hardware. We execute the experiments on a server with 4 NVIDIA Volta V100 GPUs, each with 16 GB of

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.

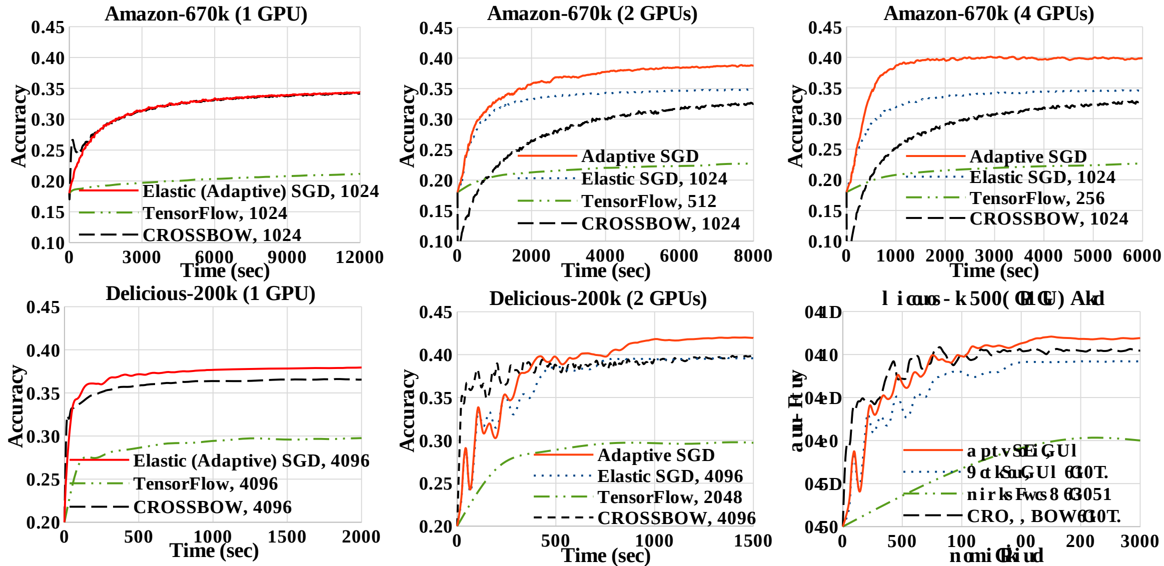


Fig. 4: Time-to-accuracy for a given number of GPUs.

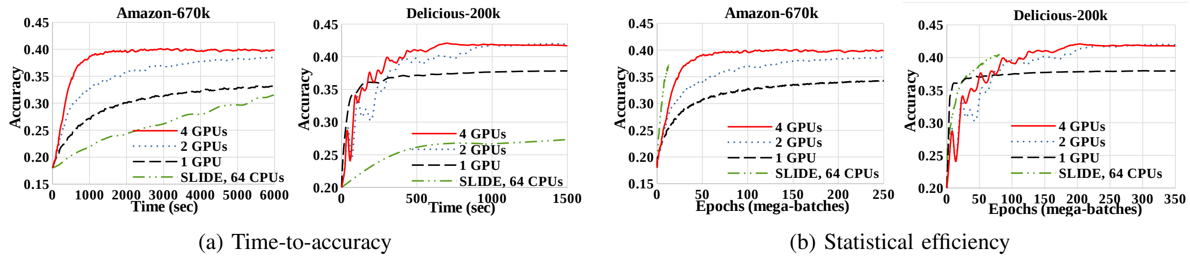
RAM. The server has a 16-core (32 threads) Intel 6226R Cascade Lake CPU with 192 GB of RAM and is running the Ubuntu 16.04.7 operating system.

B. Results

Time-to-accuracy. The time-to-accuracy measure is the most relevant metric to evaluate a training algorithm because it assesses the wall-clock time to achieve a certain level of accuracy. The shorter this time is, the less resources an algorithm uses. Figure 4 depicts time-to-accuracy for all the considered GPU methods. When the testing configuration has a single GPU, all the methods become mini-batch SGD. Since Elastic and Adaptive SGD are both implemented in HeteroGPU and use the same model update rule, they are identical—the reason they are depicted by a single curve. The model update in CROSSBOW includes the deviation of the local replica from the global model, thus, the different behavior. The TensorFlow curve is completely separate from the others because of the different implementation. There are several common trends across the two datasets. Adaptive SGD achieves the highest accuracy among all the methods in the shortest interval of time. This is the case for all the GPU configurations. The improvement over Elastic GPU proves the benefits of variable batch sizes and weighted model averaging. Without these, the delay from the slower GPUs increases the time to process a mega-batch. Moreover, the variation in batch size adds more variability to the local model replicas, which reflects in a more generalizable global model. Another trend is TensorFlow's considerably slower time-to-accuracy. There are two reasons for this. First, the execution of an SGD epoch and mirrored all-reduce aggregation are slower. Second, the global model is updated after every batch. In the case of Adaptive and Elastic SGD, the global model is updated only after a mega-batch having the size of 100 batches. CROSSBOW displays the most variability across the two datasets. It achieves much better accuracy on Delicious-200k than on Amazon-670k. This discrepancy is due to the sensitive global model update that can lead to divergent local replicas. There are two manifestations of the divergence. First, there is poor accuracy—the case for Amazon-670k. Second, there is instability—the case for Delicious-200k. While periodic replica recalibration can alleviate the divergence, the trigger and frequency of this operation are not well-defined. Nonetheless, the gap to Adaptive SGD is quite significant—especially for Amazon-670k.

Scalability. In order to analyze the training scalability of Adaptive SGD, we plot the time-to-accuracy (Figure 5a) and statistical efficiency (Figure 5b) as a function of the number of GPUs. We also include the curves for the optimized SLIDE algorithm as a CPU baseline in the figures. The benefit of using multiple GPUs is clear for the Amazon-670k dataset, where the accuracy on 4 GPUs is clearly better than on 2 and 1 GPU, respectively. This proves the importance of adaptive batch size scaling

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.



(a) Time-to-accuracy

(b) Statistical efficiency

Fig. 5: Time-to-accuracy (a) and statistical efficiency (b) comparison between Adaptive SGD and SLIDE.

and normalized model merging. On the Delicious-200k dataset, the impact made by adding more GPUs is not so straightforward. The 4 GPUs configuration provides only a limited reduction in the time to reach the maximum accuracy over 2 GPUs. Moreover, these multi-GPU configurations require more epochs to ramp-up accuracy compared to the single GPU setting. Nonetheless, 4 GPUs always achieve the highest accuracy in the shortest time interval. Compared to SLIDE, all the Adaptive SGD GPU configurations – including single GPU – are superior to the optimized CPU algorithm. This is due to the much higher hardware efficiency of the GPU over CPU since the CPU statistical efficiency is net superior (Figure 5b). The reason is the higher number of model updates. Consequently, while specialized algorithms are valuable, they cannot easily outperform adequately tuned solutions on superior computing architectures.

Do batch size scaling and perturbation activate in Adaptive SGD? Although batch size scaling and perturbed model merging are part of the Adaptive SGD algorithm, their activation is conditioned by certain triggers. Batch scaling happens only when the GPUs process a different number of replica updates, while perturbation is added to model merging only when all the replicas are well-regularized. In order to quantify how often these conditions are met during training, we plot the evolution of the batch size for every GPU in Figure 6a and the activation frequency of perturbation in Figure 6b. We observe how the batch size changes across GPUs after every mega-batch. The batch sizes are initialized with the maximum allowable value and fluctuate until they converge to a limited range in which the number of updates to the local replicas stabilize. At this point, all the GPUs perform a synchronized operation that is optimal for training. The perturbation frequency is controlled by the threshold $pert_{thr}$—set to 0.10 following an empirical study. From Figure 6b, we see that perturbation is added to merging with a very high frequency, meaning that the local replicas are well-regularized. These results confirm that the novel characteristics of Adaptive SGD are the source for its improved performance.

C. Summary

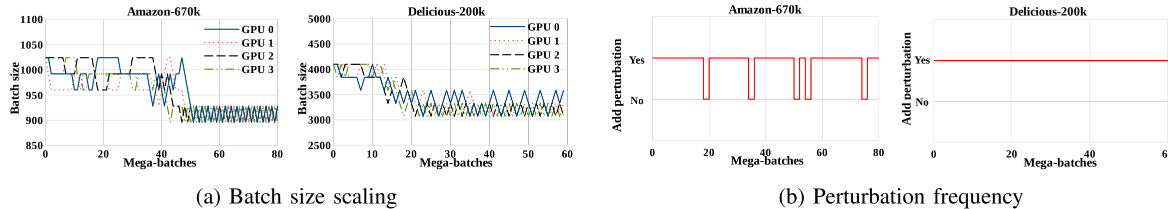
Based on the presented results, we can answer the questions raised at the beginning of this section:

- Due to the careful handling of heterogeneity – which allows a more thorough exploration of the optimization space – Adaptive SGD outperforms all the competitors in time-to-accuracy. In fact, Adaptive SGD achieves the highest accuracy among all the algorithms. In terms of statistical efficiency, SLIDE requires fewer epochs to achieve a certain level of accuracy due to the larger number of model updates performed across multiple CPU threads. However, this requires substantially longer time than Adaptive SGD.
- As we increase the number of GPUs, Adaptive SGD exhibits both faster time-to-accuracy – which is expected – as well as higher overall accuracy—which cannot be reached by any of the other methods.
- Adaptive batch size scaling and normalized model merging with perturbation – the two identifying characteristics of Adaptive SGD – are frequently invoked during execution, which confirms that they are the reason for the superior Adaptive SGD performance.

VI. Conclusions

In this paper, we introduce Adaptive SGD, an adaptive elastic model averaging stochastic gradient descent algorithm for heterogeneous multi-GPUs. Adaptive SGD addresses the challenges raised by parallel training large deep learning models on sparse data with dynamic scheduling, adaptive batch size scaling, and normalized model merging. We provide an implementation of Adaptive SGD and compare its performance against four existing methods. The results show that Adaptive SGD outperforms all the other solutions in time-to-accuracy and confirm its scalability with the number of GPUs. A

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.



(a) Batch size scaling

(b) Perturbation frequency

Fig. 6: The batch size of a GPU is adaptively scaled (a) and perturbation is frequently added to normalized model merging (b) after processing every mega-batch.

considerably extended version of the paper is available as an online technical report [16].

Acknowledgments: This work is supported by a U.S. Department of Energy Early Career Award (DOE Career).

References

[1] K. Backstrom, I. Walulya, M. Papatriantafilou, and P. Tsigas. Consistent Lock-free Parallel Stochastic Gradient Descent for Fast and Stable Convergence. In *IPDPS 2021*.

[2] K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma. The Extreme Classification Repository: Multi-label Datasets and Code. http://manikvarma.org/downloads/XC/XMLRepository.html, 2016.

[3] L. Bottou, F. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 60(2), 2018.

[4] B. Chen, T. Medini, and A. Shrivastava. SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. *CoRR*, arXiv/1903.03129, 2019.

[5] J. Chen, R. Monga, S. Bengio, and R. Józefowicz. Revisiting Distributed Synchronous SGD. *CoRR*, arXiv/1604.00981, 2016.

[6] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *EuroSys 2016*.

[7] K. Dahiya, D. Saini, A. Mittal, A. Shaw, K. Dave, A. Soni, H. Jain, S. Agarwal, and M. Varma. DeepXML: A Deep Extreme Multi-label Learning Framework Applied to Short Text Documents. In *WSDM 2021*.

[8] M. Dixon, D. Klabjan, and J. Bang. Classification-based Financial Markets Prediction using Deep Neural Networks. *CoRR*, arXiv/1603.08604, 2016.

[9] P. Goyal, L. Wesolowski, P. Dollar, A. Kyrola, R. Girshick, A. Tulloch, P. Noordhuis, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR*, arXiv/1706.02677v2, 2018.

[10] G. Hinton et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing*, 29:82–97, 2012.

[11] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS 2013*, pages 1223–1231.

[12] T. Jiang, D. Wang, L. Sun, H. Yang, Z. Zhao, and F. Zhuang. LightXML: Transformer with Dynamic Negative Sampling for High-Performance Extreme Multi-label Text Classification. In *AAAI 2021*.

[13] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch. CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB*, 12(11), 2019.

[14] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *ICML 2018*, pages 3043–3052.

[15] Y. Ma, F. Rusu, and M. Torres. Stochastic Gradient Descent on Modern Hardware: Multi-core CPU or GPU? Synchronous or Asynchronous? In *IPDPS 2019*.

[16] Y. Ma, F. Rusu, K. Wu, and A. Sim. Adaptive Elastic Training for Sparse Deep Learning on Heterogeneous Multi-GPU Servers. *CoRR*, arXiv/2110.07029, 2021.

[17] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild: A Lock-Free Approach to Parallelizing SGD. In *NIPS 2011*.

[18] C. Qin, M. Torres, and F. Rusu. Scalable Asynchronous Gradient Descent Optimization for Out-of-Core Models. *PVLDB*, 10(10):986–997, 2017.

[19] T. Ren, M. F. Modest, A. Fateev, G. Sutton, W. Zhao, and F. Rusu. Machine Learning Applied to Retrieval of Temperature and Concentration Distributions from Infrared Emission Measurements. *Applied Energy*, 252(113448), 2019.

[20] S. Ruder. An Overview of Gradient Descent Optimization Algorithms. *CoRR*, arXiv/1609.04747v2, 2017.

[21] S. Sallinen, N. Satish, M. Smelyanskiy, S. Sury, and C. Ré. High Performance Parallel Stochastic Gradient Descent in Shared Memory. In *IPDPS 2016*.

[22] S. B. Shriram, A. Garg, and P. Kulkarni. Dynamic Memory Management for GPU-based Training of Deep Neural Networks. In *IPDPS 2019*.

[23] R. You, S. Dai, Z. Zhang, H. Mamitsuka, and S. Zhu. AttentionXML: Extreme Multi-Label Text Classification with Multi-Label Attention Based RNN. In *NeurIPS 2019*.

[24] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh. Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes. In *ICLR 2020*.

[25] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. ImageNet Training in Minutes. In *ICPP 2018*.

[26] S. Zhang, A. Choromanska, and Y. LeCun. Deep Learning with Elastic Averaging SGD. In *NIPS 2015*, pages 685–693.

[27] F. Zhou and G. Cong. On the Convergence Properties of A K-step Averaging Stochastic Gradient Descent Algorithm for Nonconvex Optimization. In *IJCAI 2018*.

[28] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2021.

[29] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl, 2021.

[30] Perlmutter. https://www.nersc.gov/systems/perlmutter/, 2021.

[31] SLIDE. https://github.com/keroro824/HashingDeepLearning.

[32] TensorFlow. https://www.tensorflow.org/, 2021.

Authorized licensed use limited to: Univ of Calif Merced. Downloaded on August 25,2022 at 16:38:54 UTC from IEEE Xplore. Restrictions apply.