MARBLE: A Multi-GPU Aware Job Scheduler for Deep Learning on HPC Systems

Jingoo Han*, M. Mustafa Rafique§, Luna Xu†, Ali R. Butt*, Seung-Hwan Lim‡, Sudharshan S. Vazhkudai‡
*Virginia Tech, §Rochester Institute of Technology, †IBM Research, ‡Oak Ridge National Laboratory
*{jingoo, butta}@cs.vt.edu, §mrafique@cs.rit.edu, †xuluna@ibm.com, ‡{lims1, vazhkudaiss}@ornl.gov

***Abstract*—Deep learning (DL) has become a key tool for solving complex scientific problems. However, managing the multi-dimensional large-scale data associated with DL, especially atop extant multiple graphics processing units (GPUs) in modern supercomputers poses significant challenges. Moreover, the latest high-performance computing (HPC) architectures bring different performance trends in training throughput compared to the existing studies. Existing DL optimizations such as larger batch size and GPU locality-aware scheduling have little effect on improving DL training throughput performance due to fast CPU-to-GPU connections. Additionally, DL training on multiple GPUs scales sublinearly. Thus, simply adding more GPUs to a system is ineffective. To this end, we design MARBLE, a first-of-its-kind job scheduler, which considers the non-linear scalability of GPUs at the intra-node level to schedule an appropriate number of GPUs per node for a job. By sharing the GPU resources on a node with multiple DL jobs, MARBLE avoids low GPU utilization in current multi-GPU DL training on HPC systems. Our comprehensive evaluation in the Summit supercomputer shows that MARBLE is able to improve DL training performance by up to 48.3% compared to the popular Platform Load Sharing Facility (LSF) scheduler. Compared to the state-of-the-art of DL scheduler, Optimus, MARBLE reduces the job completion time by up to 47%.**

I. Introduction

Artificial intelligence (AI) technology using deep learning (DL) [1] is gaining rapid popularity in industry and science because of its powerful abilities for image recognition, natural languages processing, and autonomous vehicles. Especially, the scientific communities have started to benefit from DL to automate, accelerate, and drive understanding at supercomputer scales, which enables breakthroughs in scientific discovery such as analyzing climate patterns [2], modeling the universe [3], and classifying medical images [4]. To meet the increasing demands of running DL applications on supercomputers, the HPC community has started designing and deploying computing systems ready for DL workloads.

The design of the newly ranked supercomputers begins to adopt extreme parallelism and heterogeneity [5]. For example, according to the Top500 list of supercomputers from November 2019 [6], top-ranked Summit [7] is equipped with $27,648$ NVIDIA V100 graphics processing units (GPUs) [8] and each is connected to IBM POWER CPU with high-speed NVLink connections [9] making it "one of the most AI-capable machines ever constructed" [10]. In fact, five of the top ten supercomputers in 2019 are GPU-enabled HPC systems as compared to two out of the top ten supercomputers in 2017. Moreover, it is now a common practice for supercomputers to add multiple GPUs in a single node. For example, the top two ranked supercomputers, i.e., Summit and Sierra [11], deploy six and four GPUs per node, respectively. Similarly, future exascale supercomputer Frontier [12] scheduled for delivery in 2021 will include four GPUs per node. This shows an increasing trend of multi-GPU based HPC systems. However, simply adding more GPUs into a node does not guarantee linear performance improvement for DL training. We observe that multi-GPU based HPC systems show different performance trends in comparison with the existing studies [13]–[17].

Our motivation experiments in Section IV where we ran typical DL jobs with increasing number of GPUs on Summit HPC system show a training throughput gain of up to $3.3\times$ with 6 GPUs. Ideally, it should show $6\times$ improvement, achieving linear scalability. This performance gap is due to the fact that newly ranked HPC systems employ powerful GPUs (e.g., V100, P100 [18]) and the conventional schedulers are unable to fully utilize multiple GPUs to their full potential. Moreover, the recent HPC systems adopt fast CPU-to-GPU connections, e.g., NVLink, that diminish the existing optimization efforts on the batch size or GPU locality. Thus, large-batch training or GPU locality-aware scheduling does not provide notable performance improvement as the I/O overhead between CPU-to-GPU communication is alleviated substantially by the fast NVLink connections. Furthermore, we observe that multi-GPU training does not fully utilize GPUs when less computationally intensive DL models are executed on each GPU.

While the latest HPC systems [7] with multi-GPU show a different scalability trend for DL, the state-of-the-art DL job[1] schedulers [15], [21]–[23] and resource managers [24], [25] are unable to capture this non-linear scalability trend. Instead, previous studies on DL scheduling focus on GPU locality to reduce CPU-GPU transfer overhead or time slicing to speed up hyper-parameter searches. Recent efforts [15], [26] focus on cloud environment and avoid resource contention and job interference when concurrently running multiple DL jobs. However, we observe negligible job interference with the powerful GPUs, high-speed connections, enough GPU cores, and large memory in latest HPC settings. Therefore, these works do not apply to the HPC environment with high speed interconnect and powerful GPUs. Moreover, traditional HPC schedulers are designed for classical HPC jobs and are unaware of the characteristics of DL training. These schedulers are often static and they cannot adjust the allocation of GPU resources to DL jobs until training completion. They allocate a fixed number of GPUs to a training job, which remains

[1]We use the term 'DL job' to represent neural network training workloads to avoid confusion between non-HPC and HPC usages. For example, training VGG-16 [19] and ResNet-50 [20] is referred to as two DL jobs.

unchanged throughout the training process, resulting in a low GPU utilization. This approach is especially inefficient to run multiple DL jobs for a single application, which is a common practice in DL, e.g., to search for the best model for the target learning task. With the performance bottleneck of scaling up a single DL training on high-performance multi-GPU settings, this paper enables better utilization of GPU resources by multi-job scheduling on multi-GPU HPC systems.

In this paper, we propose MARBLE, a multi-GPU aware job scheduler for DL applications on modern HPC environments that aims to address the aforementioned challenges. MARBLE is a scheduling and resource management system specific for efficient DL training on HPC systems. The existing HPC schedulers cannot prevent waste of limited HPC resources where all jobs are scheduled on a single set of nodes (resource quota per user) to provide efficient resource sharing between multiple users. Within the limited resource environment, the current schedulers have to run multiple DL jobs only in a sequential way. Unlike the existing schedulers that optimize on the locality of workers, GPU locality, or resource configuration of parameter servers and workers, MARBLE incorporates the characteristic of non-linear scalability of multi-GPU at the intra-node level on modern HPC environments. To this end, MARBLE runs multiple DL jobs concurrently with dynamic GPU resource assignment via a suspend and resume mechanism. MARBLE first decides an appropriate number of GPUs according to the model scalability on a multi-GPU system for a job and executes it with the optimal number of GPUs. Then it runs other jobs in parallel on the leftover GPUs on a node. Assigning an appropriate number of GPUs enhances DL training throughput and co-locating multiple DL jobs within a single node achieves higher GPU utilization.

Specifically, this paper makes the following contributions:

- We provide insights on how DL training throughput is affected by multiple GPUs in terms of scalability on a single node and analyze why large-batch training or GPU locality is not an important factor of DL training throughput on the latest HPC systems.
- We design DL job scheduling heuristic to concurrently run multiple DL jobs on a single node using a suspend and resume mechanism. To the best of our knowledge, this is the first work to consider the non-linear scalability of GPUs per node on HPC systems for DL training.
- We implement and evaluate MARBLE with DL job sets on the world's fastest supercomputer, Summit, equipped with six V100 NVIDIA GPUs on each node connected with NVLink. We evaluate MARBLE for both single- and multi-node training. Compared to the current LSF [27] job scheduler, MARBLE reduces the total completion time of training DL jobs by up to 48.3% and increases the overall GPU utilization by up to 86%.

II. Background

A. HPC Supercomputer Architecture

HPC systems, e.g., Summit, have evolved into systems with extreme parallelism and heterogeneity by deploying multiple GPUs per node to meet the high concurrency requirements from both scientific simulations and AI workloads. However, introducing multi-GPU leads to high I/O cost between GPUs as well as the communication overhead between CPUs and GPUs. To reduce the I/O bottleneck, the latest HPC systems have begun to use NVLink [9] that is 5× faster than the conventional PCIe [28] interface. Large scale HPC systems with thousands of nodes also adopt high-performance networks between nodes, e.g., 115 TB/s bisection bandwidth [29] to avoid network bottleneck. Apart from high-performance parallel storage systems, e.g., GPFS [30] and Lustre [31], each node is typically equipped with a non-volatile memory (NVM) [32] to be used as burst buffer to speed up data I/O. An example of such an architecture is shown in Figure 1, which handles DL workloads efficiently [2], [10], [29], [33].

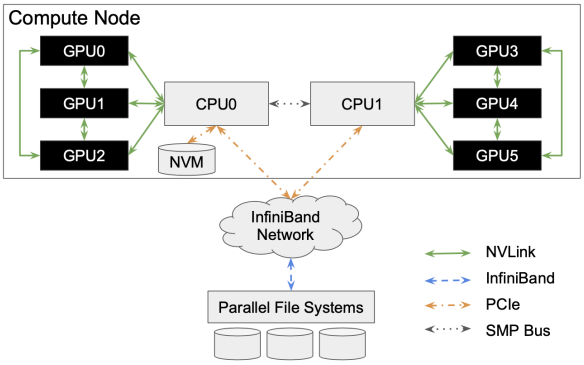


Fig. 1: System architecture of Summit.

TABLE I: CNN model information. (Conv. = Convolution, Incept. = Inceptional, FC. = Fully Connected)

CNN Model	Number of Layers			Number of Parameters
	Conv.	Incept.	FC.	
LeNet [36]	2	0	2	60K
AlexNet [37]	5	0	3	61M
ResNet-50 [20]	49	0	1	25M
Inception-v3 [38]	7	11	1	24M
ResNet-101 [20]	100	0	1	45M
VGG-16 [19]	13	0	3	138M

B. Convolutional Neural Network (CNN)

CNN is one of the DL methods that contains multiple convolutional layers. We focus on CNN because the scientific community mainly uses DL for image analysis workloads [2], [3], [33]–[35]. Each layer in CNN consists of a set of filters to extract features from the input. Convolution of feature maps requires multiplications and accumulations for all input data, hence DL training involves a series of convolution operation with a large number of parameters at each layer. Overall, CNN training requires high computational capacity to perform a series of iterative steps over a large number of data samples.

Table I shows well-known CNN models which are used as representative workloads in this paper. LeNet [36] is a classical CNN model which is relatively small and requires less computing resources. AlexNet [37] is the first implementation to use GPU for training CNN models. ResNet [20] uses residual

learning as shortcut connections to solve the degradation problem of very deep models. The Inception model [38] contains inception layers where smaller multiple convolution layers and a pooling layer reside as one or two layers, which makes computation cost much lower than the previous DL models. Although ResNet-50 and Inception-v3 contain much deeper layers than AlexNet, their parameter set is less than half of the size of AlexNet because of residual blocks and inception layers. VGG-16 [19] has a simple architecture but contains too many parameters due to three fully connected layers. These models require different computing powers and system resources based on their fundamental design choices.

III. Related Works

There are existing studies on scheduling and resource management for DL training. Optimus [21] adjusts resource allocation dynamically based on DL training time prediction using job progress. Cynthia [25] provides cost-efficient resource management via profiling network throughput. Optimus and Cynthia focus on adjusting cluster configuration (e.g., the numbers of parameter servers and worker nodes), which does not work well with HPC environment with quota based resource allocation. However, MARBLE aims to dynamically allocate GPU resources per node within the allocated cluster. Gandiva [15] manages GPU resources by leveraging GPU time sharing for mainly targeting efficient hyper-parameter search. However, it is known that Gandiva achieves limited improvement in the job completion time [22]. Tiresias [22] performs scheduling based on the number of used GPUs and executed time, which minimizes job completion times of DL training. RALP [24] places memory-intensive layers into parameter servers to reduce network traffic between parameter servers and worker nodes. Salus [39] performs fine-grained GPU sharing to reduce average job completion time. Most of these works focus on more flexible cloud environments. Compared to these works, MARBLE focuses on end-to-end application (may include multiple DL jobs with or without parameter server) run time on latest HPC systems [7], [11].

These existing efforts do not consider the non-linear scalability of multiple GPUs across the nodes for reducing DL training time. To the extent of our knowledge, this paper is the first to leverage the non-linear scalability of multi-GPU for DL training. The recent studies [15], [39] use a suspend and resume mechanism for the time-slicing scheduling for DL training, while our work uses suspend and resume for dynamically assigning optimal GPU resources to DL jobs.

IV. Motivation

In this section, we describe our observation of the current state of the art in multi-GPU scalability and the insensitivity for batch sizes and GPU locality for DL workloads, which motivates our work.

A. Non-linear Scalability of Multiple GPUs

We study the impact of the number of GPUs in a single server on the training throughput performance of different DL models and show the result in Figure 2. Unlike the existing studies [16], [17], [40], [41], training using more than 3 GPUs on a single node is not scalable in our target HPC setting. Moreover, three DL models, i.e., LeNet, ResNet-50, and VGG-16 experience performance degradation, when they are trained on 6 GPUs. However, we observe that compute-intensive workloads lead to more scalable multi-GPU training. Thus, when the computation workload is not high (e.g., LeNet), it has a negative impact on the scalability because adding more GPUs for less compute-intensive workloads results in low overall GPU utilization [42]. To verify this, we also plot GPU utilization results in Figure 3. LeNet presents a low GPU utilization and adding more GPUs causes more coordination overhead than the benefit from parallelism. Training AlexNet or ResNet-50 on two GPUs achieves substantial speedup as compared to using a single GPU. However, training using 3 or 4 GPUs does not give substantial performance improvement because of incurred communication overhead between GPUs. Conversely, the training performance of VGG-16 keeps increasing with 3, 4, or 5 GPUs. Hence, for DL models with a high computational workload, the computational workload is divided among multiple GPUs to achieve better throughput benefits than synchronization overhead.

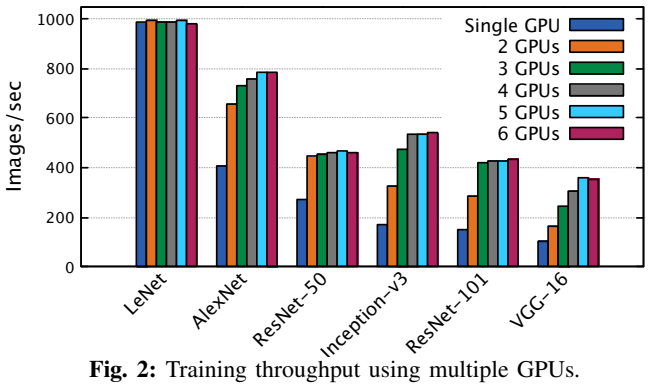


Fig. 2: Training throughput using multiple GPUs.

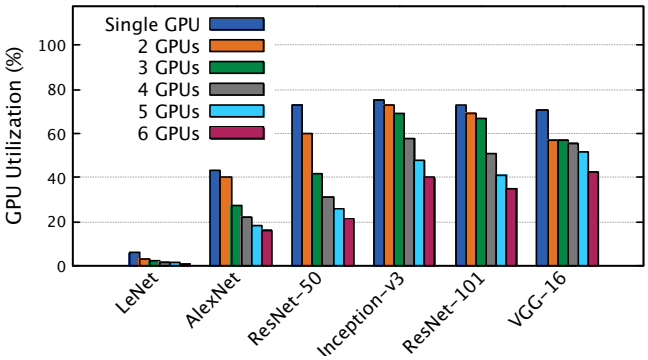


Fig. 3: GPU utilization using multiple GPUs.

The current multi-GPU environment of the latest HPC system does not provide linear scalability for DL training. Moreover, each DL model shows a different scalability trend. Therefore, the scheduler should consider an optimal number of GPUs for different DL jobs instead of simply adding more GPUs to enhance throughput and efficiency.

B. Insensitivity to Batch Size

Batch size is one of the most important configuration factors that affect DL training throughput [14], [43]. Since large-batch training decreases gradient aggregation frequency and weight update, it requires fewer iterations. Although large batch size results in low accuracy, there are efforts [2], [3], [43] to address this issue by using learning rate algorithms, such as Layer-wise Adaptive Rate Scaling (LARS) [44]. These approaches show that large-batch training gives performance benefits from large-scale HPC systems without loss of accuracy.

According to our recent study [42], increasing the batch size from 128 images to 256 images results in a slight performance enhancement, i.e., $1.0\% \sim 3.7\%$ for three models (LeNet, AlexNet, ResNet-50). However, performance degradation occurs when larger batch sizes, e.g., 4K and 8K, are used. This result shows different performance patterns from a recent work [14] which observed linear performance improvements with increasing batch size from 256 to 11K images using P100 GPUs. Previous studies [13], [45] assumed that connections between CPU and GPU are slow. However, the latest supercomputers have started employing NVLink between CPUs and GPUs, which provides $5\times$ faster CPU-to-GPU throughput than PCIe bus [28]. As a result, a large batch size for hiding communication overhead is not effective to improve performance for these systems.

C. Insensitivity to GPU Locality

We measure the impact of GPU locality within the same node. When 2 GPUs having different CPU affinity are used for a 2-GPU training job, performance degradation occurs due to the communication overhead [15]. However, we observe that GPU locality within a single node has a little impact on the training throughput. We train each DL model with two different CPU-affinity configurations, i.e., the same socket and different socket configurations. The performance difference is very slight (i.e., $0.2\% \sim 1.9\%$). This is because as stated earlier, the latest HPC systems deploy NVLink between GPUs and CPUs connections, which mitigates communication overhead. For example, the X-Bus on Summit is 64GB/s and the NVLink is 50GB/s; so a CPU can get to another CPU-GPU complex at NVLink speeds. Thus, GPU locality is no longer a significant factor for fast DL throughput, which suggests that DL scheduling policy does not need to incorporate GPU locality when allocating GPUs to DL jobs.

D. Limitation of HPC Cluster Scheduler

Several schedulers, e.g., SLURM [46], IBM Platform LSF [27], have been proposed for HPC environments for batch job processing. Unlike other cloud environments or non-HPC systems, dedicated machines are allocated exclusively for a single HPC user in a gang-scheduling fashion. Job scheduling, queuing and monitoring on HPC systems are maintained by batch schedulers [47]. Usually, these HPC schedulers target long-running jobs for scientific simulation and modeling workloads [48]. Jobs are received from users in batches. These jobs are placed into job queues and are dispatched to compute nodes when required resources are available. These gang-schedulers do not provide flexibility for dynamic configuration and resource sharing. The resource configuration is defined by the user at the time of job submission and remains fixed until job completion even if the resources are configured inefficiently and wastefully. Moreover, the current HPC schedulers do not support features optimized for DL workloads and treat DL jobs as black-boxes, i.e., they do not leverage the non-linear characteristics of multi-GPU training. To enhance system performance and utilization of emerging HPC systems, GPU resources should be allocated efficiently according to the scalability of multi-GPU workloads with dynamic resource management. However, it is difficult to change the existing HPC job schedulers as the majority of the resident applications on HPC are still modeling and simulations. To this end, MARBLE does not aim to completely replace HPC job schedulers but works alongside HPC job scheduler for DL jobs to dynamically adjust the GPU resources within the resource set allocated by HPC job schedulers.

V. Design

In this section, we describe the details of the developed job scheduler, MARBLE, for DL workloads that assigns GPU resources to each job based on multi-GPU scalability and supports dynamic allocation of GPUs during training. As we show in Section IV, a large-batch size and GPU locality have little impact on the training throughput. Therefore, MARBLE does not consider the batch size and GPU locality, and instead addresses the following:

- Reducing the overall job completion time by allocating the optimal number of GPUs based on non-linear scalability of multi-GPU training.
- Increasing GPU utilization by collocating multiple jobs in parallel with a suspend/resume mechanism.

A. Design Objectives

The objective of MARBLE is to minimize the total training time (T) of all DL jobs, i.e., $T = \sum_{\forall^i} t_i$, where t_i denotes the original training time of a DL job i without scalability-awareness. We assume that jobs are executed serially with different starting times within limited resource settings (e.g., single-node training). Each t_i can be reduced by α through allocating the optimal GPUs that provide the best throughput, instead of using the maximum number of GPUs provided by a single node. α is dependent on the DL models and HPC systems. As shown in Figure 2, the maximum value of α is 2.91% on the studied HPC systems when VGG-16 is trained on 5 GPUs instead of 6 GPUs. Additionally, the total execution time can be reduced by parallel execution of DL jobs, because multiple DL jobs can be concurrently executed on the same node. Thus, total time T can be optimized by scheduling algorithm of MARBLE as:

$$T = \sum_{\forall^i} (t_i \times (1 - \alpha_i) - p_i) \quad (1)$$

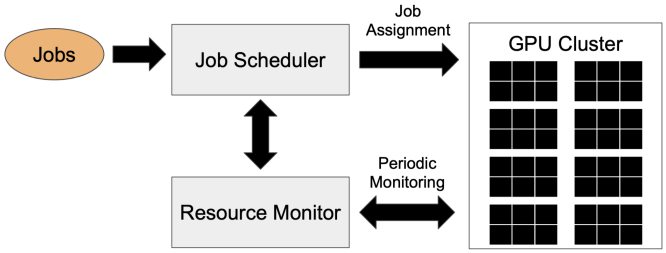


Fig. 4: MARBLE components and their interactions.

p_i indicates the gain in execution time of each DL job when a DL job is executed concurrently with other DL jobs. For example, if a DL job is executed concurrently with other DL jobs with higher priority until the completion of the DL job, p_i will be equal to $t_i \times (1 - \alpha_i)$ of the DL job. However, if all GPUs on a node are used by a single DL job, then the value of p_i will be $zero$ as no performance gains would be achieved through concurrent execution.

B. System Architecture

MARBLE is composed of a job scheduler and a resource monitor as shown in Figure 4. It resides next to the cluster-level job scheduler and takes control of the node-level DL job scheduling by reading a set of submitted DL jobs into a FIFO queue. The user does not need to consider the number of GPUs. Instead, MARBLE determines the GPU numbers automatically according to the scalability of the DL model and the availability of GPUs in the system. MARBLE then schedules a job with the optimized number of GPUs and sends these jobs to DL frameworks (e.g., TensorFlow) for execution through the existing HPC scheduler.

C. Scheduling Policy

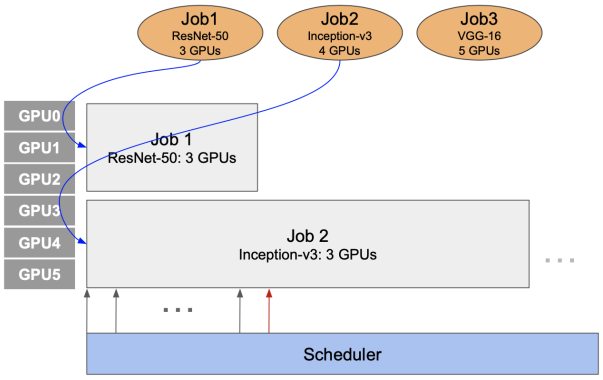
MARBLE employs a FIFO-based scheduling policy and runs multiple jobs in parallel as either a primary or a secondary job. The primary job is defined as one where the optimal number of GPUs is assigned to provide the best throughput. The secondary jobs are jobs that share GPU resources with other jobs and run using a suboptimal number of GPUs. We use the FIFO scheduler to preserve the task order (in case tasks are dependent on each other) and scheduling fairness. Algorithm 1 gives a detailed procedure of our scheduling mechanism. The scheduler picks the first job in the queue as a primary job, decides the optimal number of GPUs based on the characteristics of the model, and assigns GPUs to the primary job, where the job can be run at the best throughput. Then, the scheduler selects the next job which is marked as independent and checks the number of available GPUs that the compute node can provide. If the optimal number of GPUs for the given job is less than the number of available GPUs, then the selected job is marked as a primary job and allocated with the optimal number of GPUs. Otherwise, the job is assigned with the remaining GPUs, thus identified as a secondary job. MARBLE repeats this procedure until all GPUs are occupied or all jobs are scheduled. MARBLE proposes a suspend/resume mechanism to dynamically allocate GPU resources to the secondary job by providing an optimal number of GPUs when free GPUs are available. The scheduler monitors all running jobs to track their completion. Upon completion of a job, MARBLE searches for a running secondary job and tries to promote it to a primary job by assigning more GPUs freed by completed jobs until the job reaches the optimal number, i.e., the secondary job is suspended and resumed with the optimal number of GPUs. If there is no running secondary job, the next job in the queue is chosen for scheduling. The new job will be run as either a primary job or a secondary job according to the number of available GPUs and the optimal number of GPUs required by the job.

Algorithm 1: Job scheduling algorithm in MARBLE.

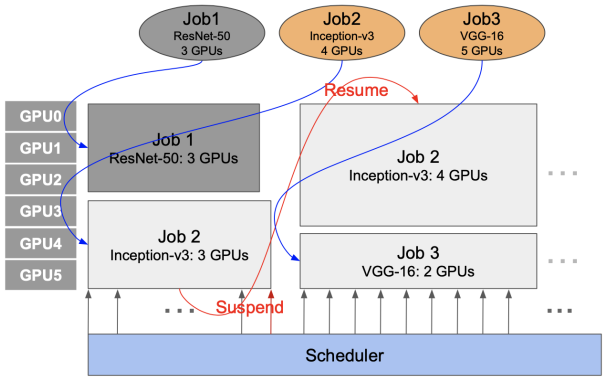
```
Input: jobSet
1  begin
2      run_multiple_jobs();
3      forall job ∈ jobSet do
4          Monitor running Jobs;
5          if Job is finished then
6              if Available GPUs >= GPU_optimal &
                Job_secondary is still running then
7                  Suspend a running Job_secondary;
8                  Resume the suspended job as Job_primary;
9              end
10             run_multiple_jobs();
11         end
12     end
13 end
14 Function run_multiple_jobs()
15     while Available GPUs > 0 do
16         if Available GPUs >= GPU_optimal then
17             Run next job as Job_primary;
18         else
19             Run next job as Job_secondary;
20         end
21     end
```

Figure 5 shows an example working of MARBLE. There are 3 jobs, i.e., ResNet-50, Inception-v3, and VGG-16 with 6 GPUs on a single node. The first job executes as a primary job as there are enough GPUs available. Thus, Job 1 (ResNet-50) is executed on 3 GPUs as training ResNet-50 on 3 GPUs is the fastest. However, Job 2 (Inception-v3) cannot be executed as a primary job because there are not enough GPUs available. Although the optimal number of GPU for training Inception-v3 is 4, Job 2 (Inception-v3) is scheduled as a secondary job with 3 GPUs. Here, for maximizing GPU utilization, when one of the jobs is finished, the current running secondary job (Job 2) on 3 GPU is suspended and resumed by MARBLE with the optimal number of GPUs (i.e., 4 GPUs). MARBLE selects Job 3 (VGG-16) as a secondary job as there are only 2 GPUs available.

The complexity of computation overhead of scheduling algorithm is $O(n)$, where n indicates the number of jobs



(a) Initial job assignment



(b) Suspend/resume during training

Fig. 5: GPU resource assignment based on multi-GPU scalability awareness.

running in parallel. The scheduling algorithm needs to choose the optimal number of GPUs and runs the jobs in parallel. For example, if 4 parallel jobs can be run after a job is finished, the algorithm will be repeated four times for allocating resources and executing jobs. Therefore, MARBLE is scalable in terms of the number of running n parallel jobs. However, the complexity of our algorithm does not depend on other components in the HPC cluster, e.g., the number of total DL jobs, the number of GPUs per node, and the number of available nodes.

D. Optimal Number of GPUs

MARBLE determines the optimal number of GPUs through offline profiling to reduce the runtime complexity. MARBLE derives the number of GPUs for each DL model from the historical execution data measured during the offline profiling stage. This information is used at the runtime at the scheduling stage to determine the optimal number of GPUs that should be assigned to run a particular job.

VI. Implementation

MARBLE is framework-independent and can work with any DL framework. However, we have implemented it with TensorFlow to leverage its checkpointing mechanism.

Suspend/Resume: We use the checkpointing mechanism for suspending and resuming jobs. The checkpointing method has been widely used for saving and restoring DL models. We checkpoint the state of the current training job and resume the job from a checkpointed file. The checkpoints are stored by TensorFlow after the completion of a DL job. The checkpoint files are saved at separate directories under the same parent directory. We use the Unix signal mechanism [49], i.e., `SIGTSTP` and `SIGUSR1` signals to terminate the running job. MARBLE sends a `SIGTSTP` signal to TensorFlow to suspend a DL job. We introduce a new flag (i.e., `SUSPEND_FLAG`) in TensorFlow to check if the signal is received from the scheduler per iteration. We also add a signal handler for a `SIGTSTP` signal to TensorFlow. When TensorFlow receives a `SIGTSTP` signal from the scheduler, the signal handler sets `SUSPEND_FLAG`. The `SUSPEND_FLAG` flag is checked by `train_step` function per iteration during training. If the `SUSPEND_FLAG` flag is set, then the `request_stop()` is called and checkpoint will be saved. After finishing checkpointing, TensorFlow sends a `SIGUSR1` signal to the scheduler, which leverages the resume operation to start the DL job with updated configuration parameters, e.g., the number of GPUs, and the number of steps

Job Monitoring: MARBLE periodically monitors all running jobs with a configurable interval, e.g., 10 seconds or 30 seconds. In our implementation, MARBLE monitors jobs by using system commands, i.e., `ls` [50] and `nvidia-smi` [51], that are not intrusive and incur minimal overhead.

VII. Evaluation

We evaluate MARBLE on a modern IBM-POWER based HPC system. We randomly selected job sets shown in Table II and Table III and show the performance of both single and multiple node training. We compare MARBLE with the default LSF scheduler used in the system. We also compare MARBLE to Optimus, a recent DL scheduler. We run experiments five times and report the averages and 95% confidence intervals. The highlights of our evaluation are as follows:

- MARBLE improves the overall job completion time by up to 48.3% and the overall GPU utilization by up to 86% for single-node training. It concurrently runs multiple DL jobs within a single server, maximizing GPU efficiency based on non-linear scalability.
- MARBLE improves the overall job completion time by up to 30.5% and the overall GPU utilization by up to 34.6% for multi-node training.

A. Methodology

Testbed. We conduct our experiments on the top-ranked supercomputer Summit. Our system architecture is shown in Figure 1. Specifically, each compute node in Summit has six NVIDIA Tesla V100 GPUs and two IBM POWER9 CPUs, 512 GB memory, one 800 GB NVMe SSD, and uses GPFS as the main storage system. The nodes are connected using Mellanox IB EDR with a 100 Gbps bandwidth switch. Each V100 GPU has 5120 CUDA cores and 16 GB memory with the memory bandwidth of 900 GB/s. POWER9 CPUs and V100 GPUs are connected by NVLink 2.0 (50 GB/s).

TABLE II: DL job sets for single-node training.

(a) Job set 1

Job ID	Network	Steps
1	ResNet-50	800
2	ResNet-50	3600
3	Inception-v3	1400

(b) Job set 2

Job ID	Network	Steps
1	Inception-v3	2400
2	ResNet-50	600
3	ResNet-101	1000

(c) Job set 3

Job ID	Network	Steps
1	ResNet-50	900
2	Inception-v3	2000
3	VGG-16	800

TABLE III: DL job sets for distributed training on multiple nodes.

(a) Job set 4

Job ID	Network	Steps
1	ResNet-50	9000
2	Inception-v3	1200
3	Inception-v3	4200
4	ResNet-50	6000
5	ResNet-101	2000

(b) Job set 5

Job ID	Network	Steps
1	Inception-v3	4800
2	Inception-v3	9000
3	VGG-16	9600
4	ResNet-50	3000
5	Inception-v3	3000

(c) Job set 6

Job ID	Network	Steps
1	ResNet-50	8400
2	VGG-16	4800
3	Inception-v3	9300
4	ResNet-50	2000
5	ResNet-101	3000

Software. We run TensorFlow 1.8 DL framework with CUDA 9.0 [52] and cuDNN 7.0.3 [53]. We built TensorFlow from the source code to run on the IBM POWER-based system.

Workloads. We select the ImageNet dataset [54] for training the model with as large data as possible in our setup. It consists of 1.2 million images of $1,000$ categories for training and $50,000$ images for validation. For training, we use TensorFlow-Slim [55] suite, a library that provides various DNN models. We choose different sets of DL jobs, as shown in Table II and Table III. For single-node training, we use 3 job sets which comprise 3 different jobs. For multi-node training, we measure 3 job sets which consist of 5 different jobs. These job sets include widely used CNN models such as ResNet, Inception-v3, and VGG-16.

Metrics. We measure the overall job set completion time as our performance metric. We also measure the GPU utilization and GPU memory utilization as our resource utilization metric. Suspend and Resume latency in seconds is provided as our runtime overhead metric.

Baseline. We compare MARBLE with IBM Platform LSF scheduler [27]. LSF is one of the widely used HPC job scheduler for large scale HPC systems [56]. LSF scheduler allocates the maximum number of GPUs (e.g., 6 GPUs) to each DL job without considering non-linear scalability. In LSF, jobs wait in a queue until there are enough GPUs available to run the job. MARBLE assigns the optimal number of GPUs to each job according to the scalability characteristics of DL models and shares GPU resources by running multiple jobs concurrently with dynamic GPU allocation.

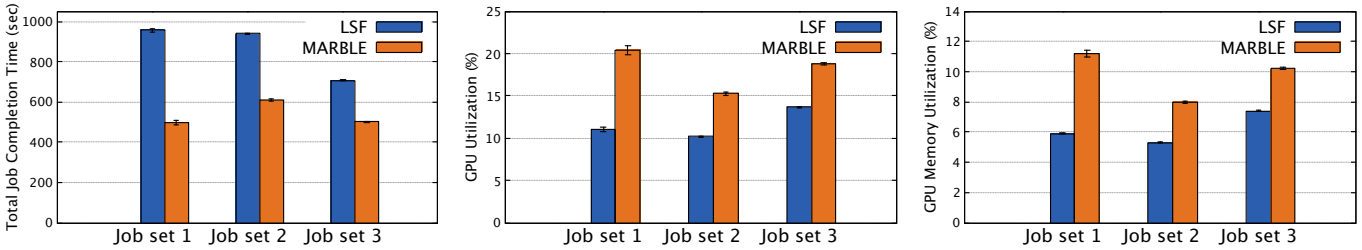
B. Performance Comparison to Baseline

In this section, we show the effectiveness of MARBLE when scheduling jobs on both single and multiple nodes. We also conduct experiments to study how scaling characteristics of the studied DL workloads impact the training throughput and GPU resource utilization.

1) Single-Node Setup: We submit the job sets shown in Table II to the system one by one, and specify to use a single node during submission. We measure the total job completion time of each set and the results are shown in Figure 6a. LSF takes 958 sec, 940 sec, and 707 sec to complete training of these 3 job sets, where all jobs use the fixed number of GPUs (e.g., 6 GPUs) during training. On the contrary, MARBLE is able to complete job sets in 496 sec, 608 sec, and 501 sec, which achieves 48.3%, 35.3%, and 29.1% reduction in the total job completion time of each job set. For job set 2, job ID 1 is selected as the primary job, where 4 GPUs are assigned because Inception-v3 shows the fastest throughput using 4 GPUs. Then, job ID 2 is executed on the remaining 2 GPUs as the secondary job in parallel with the primary job. When job ID 2 is finished, job ID 1 is not finished, yet. Then, job ID 3 is executed on the remaining GPUs as a secondary job. When job ID 1 is finished, job ID 3 is suspended by the scheduler and resumed with 3 GPUs. Job ID 2 can be completed faster than the existing scheduling approach because the job ID 2 on remaining GPUs is executed as a secondary job in parallel with job ID 1 earlier than job ID 2 is executed as a primary job. Therefore, MARBLE leverages non-linear scalability of multi-GPU with parallel job execution and reduces the overall completion time of DL training jobs.

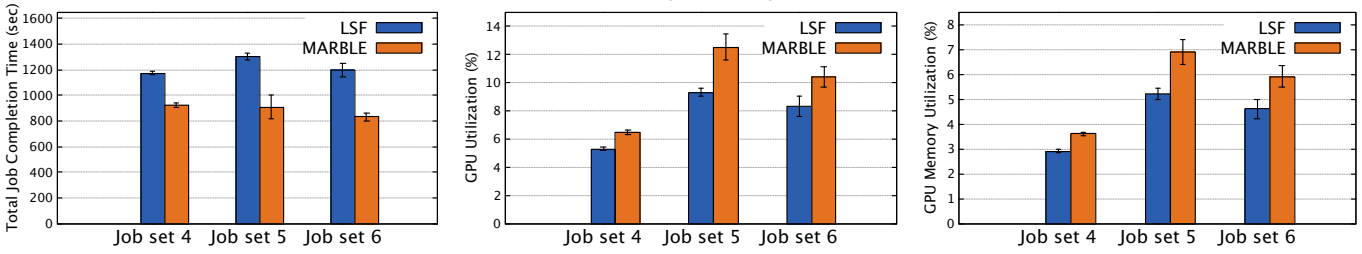
GPU Utilization: MARBLE achieves higher GPU utilization. As shown in Figure 6b and Figure 6c, GPU utilization and GPU memory utilization of job set 1 increase by 86% and 91% respectively, compared to LSF. Other job sets also provide improvements, i.e., 51% and 52%, 37% and 38% for job set 2 and job set 3, respectively. This is because two jobs are executed in parallel and the available GPUs are utilized fully through suspend/resume mechanism. As a result, MARBLE improves the overall GPU utilization, as well as the total job completion time.

2) Multi-Node Setup: We measure how MARBLE impacts distributed training on multiple nodes. To do so, we employ the parameter server architecture [57] for distributed training with data parallelism, which consists of one parameter server and eight worker nodes. Input data and computational workloads



(a) Overall job completion time. **(b)** Average GPU utilization. **(c)** Average GPU memory utilization.

Fig. 6: Training on a single node.



(a) Overall job completion time. **(b)** Average GPU utilization. **(c)** Average GPU memory utilization.

Fig. 7: Distributed training on multiple nodes.

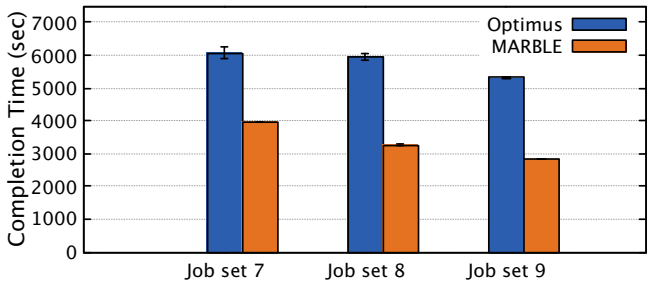


Fig. 8: Overall job completion time of Optimus and MARBLE.

are distributed across all worker nodes. Each worker node retrieves the latest parameters from the parameter server, reads input data from the storage systems, processes training, and sends updated parameters back to the parameter server. We submit job sets from Table III one by one and measure job training time, GPU utilization, and GPU memory utilization. Figure 7 shows the results. Similar to the single-node setting, MARBLE achieves performance improvement of 21.1%, 30.5%, and 30.5%, for job set 4, 5 and 6, respectively, as shown in Figure 7a. MARBLE is also able to increase GPU utilization (shown in Figure 7b) by up to 34.6%. Similar results, i.e., 22.1%, 33.0%, and 27.6% are observed for GPU memory utilization (shown in Figure 7c).

C. Performance Comparison to Other DL Scheduling

We compare MARBLE to Optimus [21]. Both approaches are based on checkpointing approach for dynamic resource allocation. While Optimus adjusts the numbers of workers and parameter servers, MARBLE configures the number of GPUs per node. As shown in Table IV, we choose different sets of DL jobs as job set 7, 8 and 9, where each DL job is trained for 1 epoch (39,936 steps). As shown in Figure 8, MARBLE improves job completion time by up to 47% as compared to Optimus. This is because, unlike Cloud, network

TABLE IV: DL job sets for Optimus.

(a) Job set 7

Job ID	Network	Steps
1	ResNet-101	39936
2	Inception-v3	39936
3	Inception-v3	39936
4	VGG-16	39936

(b) Job set 8

Job ID	Network	Steps
1	Inception-v3	39936
2	ResNet-50	39936
3	VGG-16	39936
4	ResNet-50	39936

(c) Job set 9

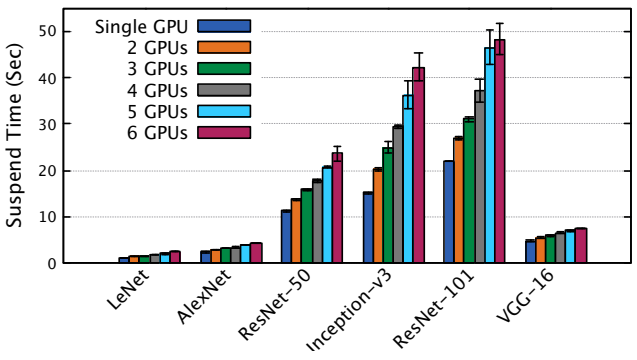
Job ID	Network	Steps
1	Inception-v3	39936
2	ResNet-50	39936
3	ResNet-50	39936
4	ResNet-101	39936

is not a significant bottleneck on the latest supercomputers. Thus, adjusting the number of GPUs per node is more effective than adjusting the numbers of Worker/PS for latest HPC systems [7], [11]. Moreover, Optimus schedules DL jobs in 10 minutes slots requiring more frequent suspend and resume than MARBLE (e.g., 9 times vs 2 times).

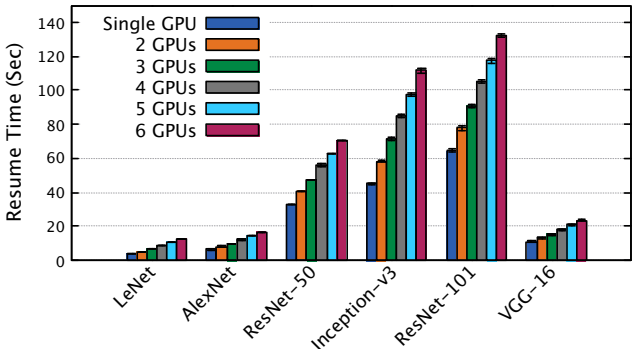
D. Runtime Overhead

MARBLE relies on runtime routines such as suspending and resuming jobs, for dynamic GPU allocation, and job and resource monitoring for instant resource scheduling, which can incur runtime overhead. In this section, we conduct experiments to evaluate the runtime overhead of MARBLE.

1) Suspend/Resume: MARBLE adopts a checkpoint-based suspend/resume mechanism to realize dynamic GPU allocation. The overhead of suspend/resume comes from file I/O on the parallel file system, including a write for suspending



(a) Suspend time.



(b) Resume time.

Fig. 9: Overhead of the suspend and resume operations.

and read for resuming. We could improve the overhead with a burst buffer. Suspend/resume time is not much variant when using multi-node, because only one node saves and restores the checkpointing file. All I/O accesses are to GPFS, therefore there is no difference in I/O performance for single and multi-node settings. However, in single-node training, MARBLE can further reduce the overhead a local bust buffer backed by an NVMe SSD. To understand the overhead of job suspension, we conduct the next set of experiments and measure the time incurred by suspending and resuming DL training of different models. The results are shown in Figure 9. The suspend and resume times are proportional to the size of the corresponding meta checkpoint file that is used for rebuilding TensorFlow graphs. For example, although the number of variables of VGG-16 is much higher than the other models (Table I), the size of meta checkpoint file of VGG-16 is much smaller than the other models such as ResNet-50, Inception-v3, and ResNet-101 (i.e., 1.31 MB vs 7.22 MB, 12.15 MB, 14.09 MB). Moreover, it shows an increasing trend with the increase in the number of GPUs as it requires additional meta-information for other GPUs. The overheads of suspend/resume are as low as 13.2% (job set 1) and up to 26.6% (job set 3) of the total training time for the job sets in Table II. Based on our results in Section VII-B, it is beneficial to suspend jobs with this overhead to achieve overall performance improvement.

2) Monitoring Overhead: We evaluate the monitoring overhead and the calculating algorithm overhead, using workloads in Table II. Here we set the monitoring interval of one second. The average monitoring time is 11.81, 11.32, and 12.66 milliseconds, respectively per monitoring cycle. The total monitoring time is 5.22, 3.79, and 4.56 seconds, which accounts for 1.05%, 0.62%, and 0.91% of the total training time respectively. This shows that MARBLE incurs negligible monitoring overhead.

3) Scheduling Overhead: We measure the scheduling overhead of MARBLE for the studied job sets. We record the scheduling overhead of 173, 124, and 112 milliseconds for the job sets in Table II. These results show that MARBLE provides overall performance improvement and increased GPU utilization with negligible runtime overhead.

VIII. Conclusion

In this paper, we present MARBLE, a multi-GPU scalability-aware job scheduling system for fast DL training, which leverages non-linear scalability characteristics of parallel training using multiple GPUs on the latest HPC platforms. Our scheduling method runs multiple DL jobs on a single node to fully utilize the GPU resources and adopts a suspend and resume mechanism to dynamically assign the optimal number of GPUs for minimizing the total execution time of the given set of DL jobs. To the best of our knowledge, this is the first effort to use non-linear scalability of multiple GPUs per node for scheduling DL training on modern HPC systems. The evaluation of MARBLE using Summit supercomputer and the representative workloads shows a performance improvement of up to 48.3%, and an improvement in the GPU utilization by up to 86%, compared to the default LSF job scheduler. In our future work, we aim to further explore how to reduce overheads of suspend/resume. Moreover, we will explore a better way of approximating an optimal number of GPUs.

Acknowledgment

This work is sponsored in part by the NSF under the grants: CCF-1919113, CNS-1405697, CNS-1615411, and CNS-1565314/1838271. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[2] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *Proc. IEEE/ACM SC*, 2018.

[3] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook, et al. Cosmoflow: using deep learning to learn the universe at scale. In *Proc. IEEE/ACM SC*, 2018.

[4] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[5] Amogh Katti, Giuseppe Di Fatta, Thomas Naughton, and Christian Engelmann. Epidemic failure detection and consensus for extreme parallelism. *SAGE IJHPCA*, 32(5):729–743, 2018.

[6] Top500. *http://www.top500.org*, 2018.

[7] ORNL Launches Summit Supercomputer. *https://www.ornl.gov/news/ornl-launches-summit-supercomputer*, 2018.

[8] NVIDIA Tesla V100. *https://www.nvidia.com/en-us/data-center/tesla-v100/*, 2018.

[9] NVLink. *https://www.nvidia.com/en-us/data-center/nvlink/*, 2018.

[10] Jonathan Hines. Stepping up to summit. *Computing in Science & Engineering*, 20(2):78–82, 2018.

[11] Sierra. *https://hpc.llnl.gov/hardware/platforms/sierra*, 2018.

[12] Frontier). *https://www.olcf.ornl.gov/frontier*, 2019.

[13] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *CoRR*, abs/1807.11205, 2018.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[15] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. USENIX OSDI*, 2018.

[16] Shaohuai Shi, Qiang Wang, Xiaowen Chu, and Bo Li. Modeling and evaluation of synchronous stochastic gradient descent in distributed deep learning on multiple gpus. *CoRR*, abs/1805.03812, 2018.

[17] Saiful A Mojumder, Marcia S Louis, Yifan Sun, Amir Kavyan Ziabari, José L Abellán, John Kim, David Kaeli, and Ajay Joshi. Profiling dnn workloads on a volta-based dgx-1 system. In *Proc. IEEE IISWC*, 2018.

[18] NVIDIA Tesla P100. *https://www.nvidia.com/en-us/data-center/tesla-p100/*, 2019.

[19] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, 2016.

[21] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[22] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.

[23] Luna Xu, Ali R Butt, Seung-Hwan Lim, and Ramakrishnan Kannan. A heterogeneity-aware task scheduler for spark. In *Proc. IEEE CLUSTER*, 2018.

[24] Jay H Park, Sunghwan Kim, Jinwon Lee, Myeongjae Jeon, and Sam H Noh. Accelerated training for cnn distributed deep learning through automatic resource-aware layer placement. *CoRR*, abs/1901.05803, 2019.

[25] Haoyue Zheng, Fei Xu, Li Chen, Zhi Zhou, and Fangming Liu. Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training. In *Proc. ACM ICPP*, 2019.

[26] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proc. ACM SOSP*, 2019.

[27] IBM Platform LSF. https://www.ibm.com/support/knowledgecenter/en/SSETD4/product_welcome_platform_lsf.html, 2019.

[28] Burak Bastem, Didem Unat, Weiqun Zhang, Ann Almgren, and John Shalf. Overlapping data transfers with computation on gpu with tiles. In *Proc. IEEE ICPP*, 2017.

[29] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proc. IEEE/ACM SC*, 2018.

[30] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. USENIX FAST*, 2002.

[31] Lustre. http://lustre.org, 2019.

[32] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE TPDS*, 27(5):1537–1550, 2016.

[33] Robert M Patton, J Travis Johnston, Steven R Young, Catherine D Schuman, Don D March, Thomas E Potok, Derek C Rose, Seung-Hwan Lim, Thomas P Karnowski, Maxim A Ziatdinov, et al. 167-pflops deep learning for electron microscopy: from learning physics to atomic manipulation. In *Proc. IEEE/ACM SC*, 2018.

[34] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. Deep learning at 15pf: supervised and semi-supervised classification for scientific data. In *Proc. IEEE/ACM SC*, 2017.

[35] Sam Ade Jacobs, Brian Van Essen, David Hysom, Jae-Seung Yeom, Tim Moon, Rushil Anirudh, Jayaraman J Thiagaranjan, Shusen Liu, Peer-Timo Bremer, Jim Gaffney, et al. Parallelizing training of deep generative models on massive scientific datasets. In *Proc. IEEE CLUSTER*, 2019.

[36] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proc. IEEE CVPR*, 2015.

[39] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[40] Víctor Campos, Francesc Sastre, Maurici Yagües, Jordi Torres, and Xavier Giró-i Nieto. Scaling a convolutional neural network for classification of adjective noun pairs with tensorflow on gpu clusters. In *Proc. IEEE/ACM CCGRID*, 2017.

[41] Shang-Xuan Zou, Chun-Yen Chen, Jui-Lin Wu, Chun-Nan Chou, Chia-Chin Tsao, Kuan-Chieh Tung, Ting-Wei Lin, Cheng-Lung Sung, and Edward Y Chang. Distributed training large-scale deep architectures. In *Proc. Springer ADMA*, 2017.

[42] Jingoo Han, Luna Xu, M Mustafa Rafique, Ali R Butt, and Seung-Hwan Lim. A quantitative study of deep learning training on heterogeneous supercomputers. In *Proc. IEEE CLUSTER*, 2019.

[43] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proc. ACM ICPP*, 2018.

[44] Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. *CoRR*, abs/1708.03888, 2017.

[45] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: training resnet-50 on imagenet in 15 minutes. *CoRR*, abs/1711.04325, 2017.

[46] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Proc. Springer JSSPP*, 2003.

[47] Rafael Dolezal, Vladimir Sobeslav, Ondrej Hornig, Ladislav Balik, Jan Korabecny, and Kamil Kuca. Hpc cloud technologies for virtual screening in drug discovery. In *Proc. Springer ACIIDS*, 2015.

[48] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scalable system scheduling for hpc and big data. *Elsevier JPDC*, 111:76–92, 2018.

[49] Maurice J Bach et al. *The design of the UNIX operating system*, volume 5. Prentice-Hall Englewood Cliffs, NJ, 1986.

[50] Unix ls. *https://en.wikipedia.org/wiki/Ls*, 2019.

[51] NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface, 2019.

[52] Compute Unified Device Architecture (CUDA). *https://developer.nvidia.com/cuda-zone*, 2019.

[53] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[54] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc IEEE CVPR*, 2009.

[55] TensorFlow-Slim. https://github.com/tensorflow/models/tree/master/research/slim, 2019.

[56] Xiuqiao Li, Nan Qi, Yuanyuan He, and Bill McMillan. Practical resource usage prediction method for large memory jobs in hpc clusters. In David Abramson and Bronis R. de Supinski, editors, *Lecture Notes in Computer Science, Springer SCFA*, 2019.

[57] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Proc. NIPS*, 2012.