SCHEDTUNE: A Heterogeneity-Aware GPU Scheduler for Deep Learning

Hadeel Albahar[∨], Shruti Dongare[∨], Yanlin Du[∨], Nannan Zhao[†], Arnab K. Paul[ρ], Ali R. Butt[∨]
[∨]*Virginia Tech, USA* [ρ]*BITS Pilani, K K Birla Goa Campus, India*
[†]*Northwestern Polytechnical University, China*
{hadeel89, dshruti20, dourlin, butta}@vt.edu, nannanzhao@nwpu.edu.cn, arnabp@goa.bits-pilani.ac.in

***Abstract*—Modern cluster management systems, such as Kubernetes, support heterogeneous workloads and resources. However, existing resource schedulers in these systems do not differentiate between heterogeneous GPU resources—which are becoming a norm—and do not support GPU sharing—which is necessary to support emerging collocation of jobs and multi-tenant applications. Thus the systems suffer from low GPU resource utilization, higher queuing delays, and an increase in application *makespan*, i.e., the duration between the arrival of the first job and the completion of the last job of a workflow. This is especially a problem in supporting crucial deep learning (DL) applications. To this end, in this paper, we profile and analyze DL jobs on heterogeneous GPUs, investigate the interference caused by collocating jobs on GPUs, and use this information to predict the GPU memory demand and job completion times. We propose SCHEDTUNE, a machine-learning-based heterogeneity-aware scheduler that ensures higher GPU memory utilization and reduced out-of-memory (OOM) failures, while supporting improved makespan. Our evaluation shows that SCHEDTUNE GPU memory predictors and scheduler outperform the state-of-the-art predictors by achieving 81% higher GPU memory utilization, 100% detection and avoidance of OOM errors, and 17.5% reduction in makespan compared to the default Kubernetes scheduler.**

***Index Terms*—Deep learning, Kubernetes, GPU sharing, Resource heterogeneity, Resource scheduling**

I. INTRODUCTION

We are witnessing rapid advancements across different aspects of machine learning (ML) and deep learning (DL), especially an exponential growth in the number and types of applications, scale, and utilization of novel architectures such as compute accelerators. Increasingly, DL applications are trained in the cloud on multi-tenant GPU clusters [24]. However, the use of on-premises self-managed clusters continues to be significant: 35% of Kubernetes users use self-managed Kubernetes clusters according to a recent report by RedHat [41]. These self-managed systems utilize cutting-edge systems (e.g., container management systems) and accelerators (e.g., GPUs) with varying computation and memory specifications. Thus, there is a growing need to manage and schedule heterogeneous GPU resources in support of containerized-DL/ML applications.

Container cluster orchestrators are integral to managing and scheduling containers on any CPU and GPU resources. The rise of containerization has led to the development of various container cluster management systems, e.g., Kubernetes, Red Hat OpenShift [40], and Apache Mesos [44]. Moreover, pre-existing cluster managers have also added support for containerized applications, e.g., Docker atop Apache YARN [5]. These platforms are evolving and supporting new workloads and using novel resources. This in turn brings a new wave of users with more applications and complex requirements, e.g., condensing DL applications to run on one or very few resources. This growing popularity and application/resources heterogeneity is furthering the gap between the capabilities of the platforms and the needs of the users.

Existing container orchestrators are typically intended for use by savvy developers who better understand their jobs requirements and can request resources that ensure time and cost efficiency. In reality, in an effort to avoid the high cost or to reduce training time, many DL users either under-provision or over-provision resources, especially GPUs [24]. Such unregulated provisioning leads to out-of-memory (OOM) errors (accounting for 8.8% of total job failures [55]) or underutilized resources [17], [24], and subsequently cause high job queuing delays, a high makespan (i.e., the elapsed time between the start and finish of a set of jobs), and a high average job completion time (JCT). The developers' tendency to request more GPU resources not only leads to provider-side under-utilization, it also degrades training performance for other users [15]. Exacerbating the problem is a recent observation that around 13.5% of jobs submitted to a GPU cluster are killed by users, wasting about 38% of GPU time [24]. Researchers are beginning to address these problems and have proposed solutions such as assigning system-determined resource shares to jobs based on available resources [22]. However, to efficiently mitigate the above problems, we need a system management-level and GPU heterogeneity-aware resource allocation solution, which can learn system behavior and closely approximate and predict GPU memory and utilization demand and manage resources accordingly.

The challenge lies in predicting the actual workload or job requirements, specifically GPU memory demand and GPU core utilization demand. In this context, multiple recent works have looked at profiling jobs. Allox [31] performs online profiling of incoming jobs to infer resource requirements by running a small representative sampling job. Gavel [34] estimates jobs' performance by mapping them to similar previously profiled jobs. DNNMem is a recently proposed

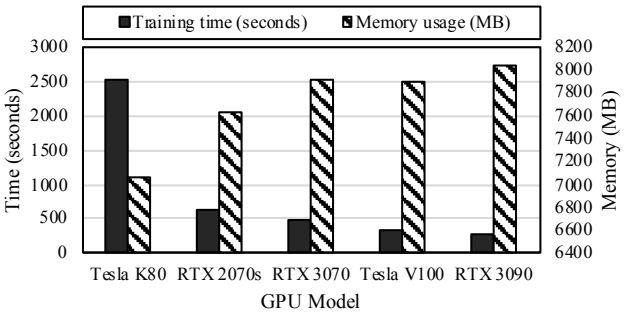

Fig. 1. GPU memory usage and training time for 1 epoch of ResNet152 [16] on Cifar10 [26] with 32 as the batch size on different GPUs. See Table I for GPU characteristics.

proprietary tool that estimates GPU memory usage of DL models [12]. Horus [54] proposes a linear equation to estimate memory required by a DL job by considering different DL model hyper-parameters.

A crucial shortcoming of prior works is that they do not explicitly consider heterogeneous GPU clusters, neither do they consider the different characteristics of heterogeneous GPU resources. We hypothesize that predicting GPU memory usage does not only depend on DL model characteristics, but also on the GPU running the job, and whether it is an inference job or a training job. To show this dependence, we trained ResNet152 [16] on Cifar10 [26] with a batch size of 32 images on five different GPUs for 1 epoch. We measured the training time and the peak GPU memory demand[1]. As seen in Figure 1, the memory demand and the training time vary by GPU model. This variance is due to several reasons. When a framework (i.e., PyTorch) initializes, the CUDA context (i.e., pre-allocated GPU memory) for an application running on a GPU differs from one GPU architecture to another and is influenced by the streaming multiprocessors (SM) count [36]. Wang et al. [48] report that the performance of single-node single-GPU workloads is affected by the GPU memory bandwidth. Moreover, Gao et al. [12] report that GPU-specific memory allocations due to ephemeral tensors and resident buffers also impact GPU memory usage. According to Amaris et al. [4] application performance (e.g., execution time) is impacted by GPU characteristics. The measurements in Figure 1 show that GPU characteristics in Table I significantly affect the GPU memory demand and job completion time, thus supporting our hypothesis.

In this paper, we first profile and analyze DL jobs training performance on different GPUs. Next, we leverage the characteristics of the DL models (e.g., input, parameters, activations, etc.) and the characteristics of GPUs (e.g., CUDA cores, Memory bandwidth, SM count, etc.) to train regression models to estimate the memory demand of DL training and inference jobs. This information is then leveraged to support GPU sharing (i.e., collocating jobs on a GPU) and to avoid performance-degrading OOM errors. Our goal is to answer the following research questions.

1) Given a job, how much GPU memory should be allocated to it without wasting resources and incurring OOM errors?
2) What job/GPU characteristics impact memory demand and JCT?
3) What is the impact of GPU sharing on JCT, i.e., how does interference impact JCT? Can we quantify this impact?

Specifically, we make the following contributions in this paper:

- We characterize performance, measure interference, and analyze single-node single-GPU DL training jobs and inference jobs on different GPU models.
- We use an ML-based approach to identify GPU characteristics that impact the GPU memory demand and JCT time. We utilize these characteristics to train GPU memory demand and JCT prediction models.
- We apply our proposed GPU memory demand prediction models, our JCT prediction models, and our interference analysis findings in the design of a GPU heterogeneity-aware scheduler, SCHEDTUNE. The scheduler improves overall efficiency of the system in a workload- and resource-aware manner.

We evaluate SCHEDTUNE by comparing it to state-of-the-art predictors, such as Horus [54] and Torchinfo [46]. We find that our prediction model achieves the lowest RMSE value and lowest average error (%). Our evaluation of SCHEDTUNE shows that SCHEDTUNE outperforms FIFO (default Kubernetes) scheduler, and achieves the lowest JCTs, higher GPU memory utilization, reduced job queuing time, and lowest workload makespan.

TABLE I
THE GPUS CONSIDERED IN OUR VARIOUS TESTS AND EVALUATION, AND THEIR KEY CHARACTERISTICS. MCS IS MEMORY CLOCK SPEED AND MBW IS MEMORY BANDWIDTH. THE GPU MEMORY IS GIVEN IN BRACKETS BESIDE EVERY GPU MODEL.

GPU	CUDA cores	Tensor cores	MBW (GB/s)	SMs	MCS (MHz)
Tesla K80 (12GB)	2496	-	240.6	13	5012
RTX 2070s (8GB)	2560	320	448	40	14000
RTX 3070 (8GB)	5888	184	512	46	16000
Tesla V100 (16GB)	5120	640	900	80	1752
RTX 3090 (24GB)	10496	328	935.8	82	19495

II. BACKGROUND

The key technology that we use and build upon is Kubernetes [27], which is a widely used open-source container management system. Kubernetes offers powerful features such as handling all aspects of system support for applications such as provisioning and deployment, scheduling and resource allocation, scaling, networking, storage, and load balancing [29]. Moreover, Kubernetes is supported by most if not all cloud providers such as Amazon Elastic Kubernetes Service [6], Azure Kubernetes Service [33], Google Kubernetes Engine [13], IBM Cloud Kubernetes Service [23], Alibaba

[1]We use nvidia-smi [35] at 250 ms intervals during the job runtime to measure GPU memory usage. We consider the maximum observed used memory value as the required GPU memory.

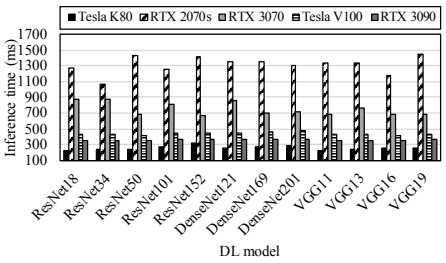

Fig. 2. Observed inference time of various DL models on five different GPUs.

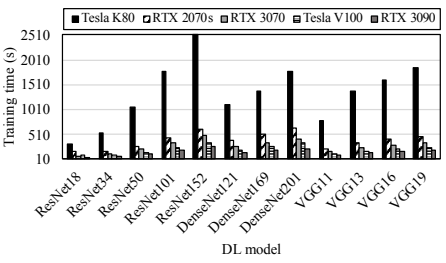

Fig. 3. Observed training time of various DL models on five different GPUs.

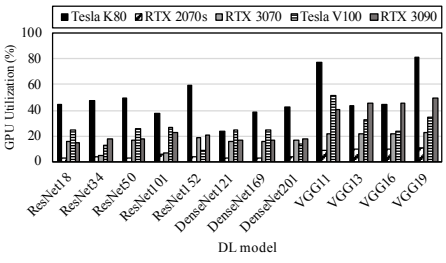

Fig. 4. GPU utilization of inference jobs of various DL models running on five different GPUs.

Container Service for Kubernetes [2], and HUAWEI Cloud Container Engine [20]). Kubernetes requires user input to specify Pod/Job resource requirements. Although the system is capable of scheduling AMD and NVIDIA GPUs to Pods/Jobs, Kubernetes allocates only full GPU cards to Pods and does not support finer-grained allocation. Moreover, the default scheduler does not differentiate between various types of GPU models and architectures. If a user wants to run a Pod/Job on a specific GPU node in a Kubernetes cluster, the user must label those nodes uniquely using Node Labels. Then the user must use Node Selectors in the Pod/Job manifest file and specify the label of the requested node so that the scheduler knows which node to assign [28]. The process is thus error-prone and cumbersome and creates a high barrier-to-entry for employing heterogeneous resources.

Kubernetes allows developers to design and implement their preferred schedulers and deploy them into their clusters. The system also supports running multiple schedulers simultaneously for individual Pod/Job. This can help to assign a Pod/Job to a appropriately feasible node, i.e., a node that meets the scheduling requirements for a Pod as defined by that scheduling algorithm.

The Kubernetes scheduler requires users to configure their job's resource requirements to assign DL Pods/Jobs to GPU nodes. For example, a Pod running a machine learning workload will be scheduled on CPU resources unless the user specifies in the Pod/Job manifest that they require GPU resources. The downside to this is that the GPU allocation task is essentially offloaded to the user. While this approach offers flexibility, not all users know how best to use the GPUs, and they certainly cannot make globally optimal allocation as a user only knows about their own workloads. Experienced developers may know the requirements of their jobs mainly after experiencing OOM errors. In that case, they usually either reduce their training batch size to fit in the GPU memory they have or train on a GPU with higher memory capacity. Most users learn by trial and error and that not only wastes time but also reduces the effective cluster GPU utilization. In contrast, new or short-term users do not even have the tool of trial-and-error and end up selecting wasteful and inefficient GPU allocations.

Running DL jobs efficiently on kubernetes requires significant fine-grained user involvement. While that fine-grained involvement is sometimes preferred, most users, whether in cloud or on-premises, would rather have the kubernetes platform itself automatically detect the cost-efficient and time-efficient resource requirements of their submitted workload which also saves users some configuration time. Having the platform automatically detect and assign resources will also benefit the cluster maintainer. Specifically, the maintainer will have a targeted finer control on resource allocation. Its target objective would be higher GPU utilization, faster response time and reduced queuing time for tenants.

III. Motivation

To motivate the need for SchedTune, we conduct a study using a set of representative DL workloads. We run training and inference jobs for computer vision models namely ResNets [16], DenseNets [19], and VGGs [43] on five different GPU nodes (Table I). We train the models on the Cifar10 [26] dataset using a batch size of 32 images for one epoch, and use models that are pretrained on the ImageNet [11] dataset for inference jobs. We request one GPU device in the Job manifest of every job and run jobs sequentially. Figure 2 shows the observed inference time. We see that, for each job, there is a lot of variance in the average inference time across the GPUs. For instance, inference time for ResNet152 varies between 312 and 1408 milliseconds. Similarly, Figure 3 shows that the training time also vary significantly. For instance, training ResNet152 can take anytime between 280 and 2513 seconds. This observed variance means that the GPU on which the job is running impacts the completion time of the job. This also means that the choice of GPU to run a DL job not only affects the job completion time, it also affects the overall cluster job throughput and queuing delays. This is because in a scenario where all the incoming DL training jobs end up assigned to GPUs that result in highest training time, the cluster throughput will be lower than the scenario when every job is assigned the available GPU that results in the lowest training time. This variance can also slow down workflows where jobs are dependent on completion of earlier tasks.

Next, we study the resource utilization of inference and training jobs. Figures 4 and 5 show the GPU[2] and memory utilization of inference workloads, respectively. We see that

[2]We use nvidia-smi [35] at 250 ms intervals during the job runtime to measure GPU utilization. We consider the maximum observed GPU utilization value as the overall GPU utilization for the job.

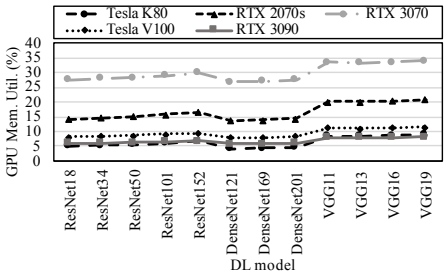

Fig. 5. Memory utilization of inference jobs of DL models running on five different GPUs.

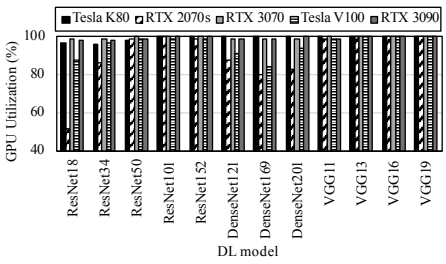

Fig. 6. GPU utilization of training jobs of various DL models running on five different GPUs.

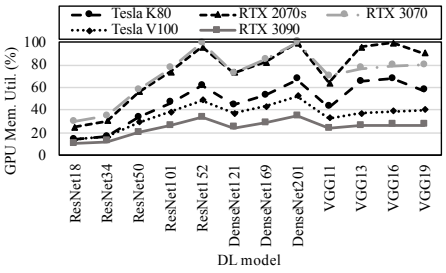

Fig. 7. Memory utilization of training jobs of various DL models running on five different GPUs.

the inference workloads exhibit significantly lower GPU utilization compared to training jobs. Moreover, the inference jobs only utilize 470 MB — 2722 MB of GPU memory across all different studied GPU types. In contrast, Figures 6 and 7 show the GPU and memory utilization, respectively, for training jobs. We note that training is a compute-intensive task, e.g., 100% GPU utilization on RTX 3090. Thus, we see that inference workloads can be collocated on a GPU to improve overall utilization, but doing the same for training jobs is not straightforward and requires further investigation.

We also repeat the experiments with a batch size of 64 images. In this case, we ran into many jobs exiting with an OOM error. For example, training ResNet152 on Cifar10 with 64 as the batch size resulted in OOM error on Tesla K80, RTX 2070s, and RTX 3070 GPUs, but successful completion on Tesla V100 and RTX 3090. This is because the memory capacity of Tesla K80, RTX 2070s, and RTX 3070 GPUs is not enough to fit the bigger batch size. Moreover, although both RTX 2070s and 3070 have 8 GB memory, we observed different outcomes when training VGG16 on Cifar10 with batch size of 64 images on RTX 2070s and RTX 3070. The observation that the training job experiences OOM error on RTX 2070s while successfully completing on RTX 3070 leads to the conclusion that the GPU memory capacity is not the only indicator for possible OOM errors. Thus, we must consider the effect of other GPU characteristics on the memory requirement of a job.

These experiments show that the variation in performance is not directly associated to a specific GPU characteristic such as the number of tensor cores, the memory clock speed (MHz), the memory bandwidth (GB/s), the number of CUDA cores, and the number of streaming multiprocessors (SMs), rather a more complex relationship exists and therefore there is a need to model such a behavior.

Finally, we test how GPU-sharing across different DL jobs impacts performance. We utilized the Volcano [9] scheduler and Volcano NVIDIA device plugin [21] that provide GPU sharing capabilities. In each of the aforementioned training jobs manifest, we request the amount of memory the job requires when it solely runs on the corresponding GPU. We submit the jobs at the same time. Figure 8 shows the minimum, maximum, and average increase in training time experienced by the collocated jobs. We noticed that each model takes

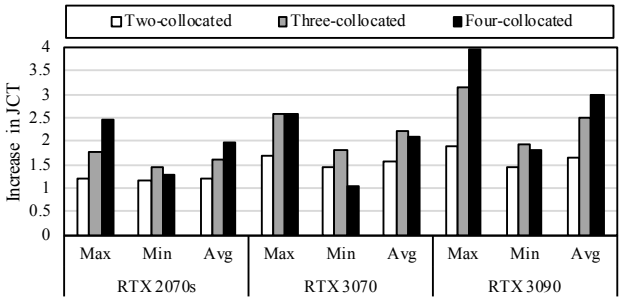

Fig. 8. Observed maximum, minimum, and average increase in job completion times when two, three, and four DL jobs are collocated on a GPU. We use ResNet18, ResNet34, ResNet50, and DenseNet121 to train on Cifar10 with 8 as batch size for 1 epoch.

longer to train when the number of collocated training jobs increase. However, the time the collocated jobs take until they all complete is less than the time they take when they run sequentially, each on a full GPU card. In this context, interference is defined as the increase in job completion time when running multiple jobs on a single GPU [47], [52], [54]. We observe that the interference is dependent on the number of collocated jobs and varies by the GPU model, but there is no direct pattern for inferring model-model interference.

Next, we run 528 combinations of all the different DL jobs in our workload on RTX 2070s. The combinations of jobs are created such that the sum total of the measured GPU memory usage of each job in the combination does not exceed the GPU memory, i.e., 8 GB. This is to avoid OOM. However, we did not identify any significant associations that allow us to quantify interference across different GPUs.

Overall, these experiments highlight the variance and difficulty in predicting GPU memory usage and collocation conditions, and show the need for centralized cluster-level control of GPU node assignments.

IV. Design

In this section, we present the design of SchedTune, a DL training and inference workloads scheduler for heterogeneous GPU clusters. We employ an ML-based approach to schedule GPU resources. We develop models to predict workload GPU memory requirements and training/inference times. SchedTune uses these predictions to assign GPUs to workloads so as to yield improved completion times. SchedTune focuses

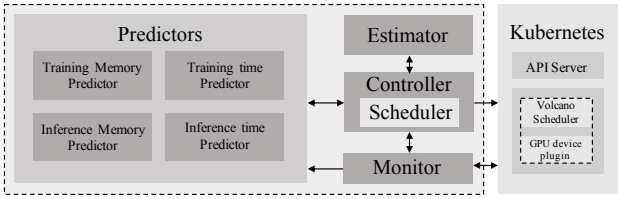

Fig. 9. The architecture of SCHEDTUNE and components therein.

on single-node single-GPU training jobs that account for up to 82% of datacenter workloads and are mostly the cause of low GPU utilization [17], [48], however, its approach can be adopted for future distributed settings.

The *objectives* of SCHEDTUNE include increasing the GPU cluster throughput, reducing workload makespan, i.e., the time elapsed from running the first job until the completion of the last job, and eliminating OOM errors. SCHEDTUNE achieves these goals by utilizing GPU sharing along with memory and training/inference time predictions, and greedily assigning most suitable GPUs to tasks.

A. SCHEDTUNE *System Architecture*

Figure 9 shows an overview of SCHEDTUNE architecture comprising four main components, namely, GPU memory and training/inference time prediction models, controller/scheduler, estimator, and monitor. Note that we build our solution atop Volcano-equipped Kubernetes [9], which provides support for simultaneously sharing GPUs across multiple jobs.

The *prediction models* are initially trained offline (Section IV-B), but can then be periodically fine-tuned as the system processes future jobs and as new GPU devices are added to the cluster. The *controller* is initialized with the characteristics of the different GPU nodes in the cluster. The controller then extracts the job characteristics (Section IV-D), processes the user-submitted manifest and passes the extracted DL model data and predictions to the scheduler. The *scheduler* receives the original user manifest and predictions, and leverages the *estimator* to create a modified manifest that now requests better/optimal GPU node and the predicted amount of GPU memory on that node (Section IV-E). The GPU memory *estimator* estimates the memory demand of the incoming jobs on the different available GPU resources and predicts needed amounts to avoid OOM errors (Section IV-C).

B. GPU Memory and Time Prediction

To seamlessly facilitate GPU sharing and job collocation, we need to determine the amount of GPU memory that jobs will require to avoid OOM errors in general, and especially when collocating jobs on the same GPU. We formulate the problem as designing a memory prediction model that can accurately produce a memory estimate for an input DL model with user-defined configurations such as batch size and given GPU resources specifications.

Since we have observed a positive correlation between model size and GPU memory utilization, we identify the following DL model characteristics as factors affecting jobs memory requirement:

$$M \propto P, A, B, I$$

Where, P refers to parameters size, A refers to activations size, B refers to batch size, and I refers to input size. The parameters are the learnable values in the DL model such as weights and biases. The activations are activation functions outputs. The batch size is user configurable. The input size is defined by the DL model. DL frameworks such as PyTorch [37] and TensorFlow [1] provide tools, e.g., `Torchinfo` [46], for detailed summaries and visual representation of the network elements of the models. We utilize these tools to extract the parameter size (MB), activations size (MB) and input size (MB) of the DL model and keep a local copy for direct and fast access.

We utilize machine learning models, specifically, nonlinear regressors that learn to capture data correlations given a sufficient training dataset. We design the training dataset by considering the two identified set of factors, the memory affecting factors of the model along with GPU device characteristics. We study several non-linear regression models from scikit-learn [38] such as Random Forest Regressor (RFR), Gradient Boosting Regressor, Voting Regressor, XGBoost for Regression, and Polynomial prediction. To avoid the possible overfitting problem caused by comparatively small datasets, we choose the simplest model, RFR, which provided best results in our initial experimentation. To obtain accurate predictions, we resorted to omitting outliers.

We were able to determine the characteristics that have significant effects on memory usage during training. These GPU characteristics are number of tensor cores, memory clock speed (MHz), memory bandwidth (GB/s), and streaming multiprocessors (SM) count. Tensor cores are a special feature in GPU devices that lead to developing kernels that use more memory to speed up neural networks. Memory clock speed (MHz) affects the allocation and deallocation policy and thereby contributes to the memory estimation. Memory bandwidth (GB/s) has long been understood as an important factor in processor performance. Streaming multiprocessors hold the executing units (i.e., CUDA cores). Similar to predicting memory needs, we are able to identify GPU characteristics that play a significant role in predicting inference and training time as well.

To create the dataset on-which the prediction models are trained, we perform offline profiling to measure the actual GPU memory requirement and training/inference time of DL jobs across five different GPU models. We populate the training dataset with data points following this structure: [Activations size (MB), parameters size (MB), input size (MB), Memory bandwidth (GB/s), CUDA cores, SM count, Memory clock speed (MHz), Tensor Cores, Target]. The Target is either GPU memory requirement or training/inference time. We omit the input size when training inference job datasets for inference predictions. For jobs with OOM error outcomes, because they are unsuccessful runs, the measured memory

used up to the OOM failure is inaccurate and will corrupt the predictor's training dataset. Instead, we utilize the memory usage pattern for a job across the other GPUs and interpolate the memory values that are missing due to OOM error. The interpolation values for memory usage are higher than the corresponding GPU memory, hence the OOM failure.

When new GPUs are added to the cluster and/or new DL models are being trained, the scheduler allocates the GPU with highest memory capacity to avoid OOM errors. Then, the monitor updates the prediction datasets accordingly. Once this happens, the prediction models can be fine-tuned at time intervals defined by the cluster maintainer.

C. GPU Memory Estimation

To detect OOM failures, we evaluate predictions obtained from the prediction model with the memory capacity of the target GPU. The predicted values are restructured according to the model's maximum error (%) and then compared with the actual memory capacity of the target GPU. Our tests show this approach to be effective and provides 100% accuracy in detecting OOM failures. Consider the following example, if the DL model to be trained is VGG19, the batch size is 64, the target GPU is RTX 2070s (7982 MB RAM), and we know the job failed due to OOM error. The prediction model will return 11140 MB as predicted memory and 8.98% as maximum error percentage. Then, the estimate is the sum of the predicted memory and the maximum error, 12140 MB. Here, since the target GPU's memory capacity is 7982 MB, the estimator predicts an OOM error.

D. Controller

Prior to operating, the controller is fed a list of different GPU devices in the cluster along with the following characteristics: the number of tensor cores, the memory clock speed (MHz), the memory bandwidth (GB/s), the number of CUDA cores, and the number of streaming multiprocessors (SMs). Ideally, we should be able to extract this information for a particular GPU either via system characteristics queries or by crawling the GPU specification website. However, this is currently not possible as websites such as Nvidia GPUs specs do not contain organized data that can be scrapped and commands such as `nvidia-smi -q` do not provide all the information. Therefore, for now we provide this information to the controller.

The controller first extracts the job name and DL model from the job manifest. Then it extracts the parameters size and the activations size from the DL model of inference jobs, and extracts the parameters size, activations size, input size and batch size from the DL model of training jobs. The parameters, activations, and input sizes are either extracted from a local copy (e.g., from the prediction models' training dataset) if the model is previously seen. Otherwise, the controller utilizes summary tools such as `Torchinfo` [46] to determine this information. Torchinfo takes as input the model name, input dimensions, and batch size. We use the default input dimensions for every model, e.g., $224 \times 224 \times 3$ for ResNets, and use 1 as the batch size. We multiply the resulting input size (MB) by the batch size and use the parameters size (MB) as-is (see Figure 10). The activations are the sum of the multiplied elements of the output shapes of every layer in the model (i.e., one forward pass/propagation). We multiply the activations by 4 bytes (for each floating point value), then we convert to megabytes to have the activations size (MB). We do not use the activations labeled forward/backward pass size that is provided by Torchinfo. This is because we consider it an overestimate since memory reallocation and memory sharing are not considered [12]. From our memory prediction design experiments and insights from DNNMem [12], we learned that considering the memory usage by both passes separately results in a higher prediction error compared to considering one pass.

```
----------------------------------------------------------
 Layer (type)               Output Shape          Param #
==========================================================
     Conv2d-1          [1, 64, 112, 112]            9,408
BatchNorm2d-2          [1, 64, 112, 112]              128
       ReLU-3          [1, 64, 112, 112]                0
  MaxPool2d-4            [1, 64, 56, 56]                0
     Conv2d-5            [1, 64, 56, 56]           36,864
      ...                      ...                   ...
----------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 62.79
Params size (MB): 44.59
Estimated Total Size (MB): 107.96
----------------------------------------
```

Fig. 10. ResNet18 summary results using Torchinfo [46].

The controller then passes these extracted data to the GPU memory prediction models to return a prediction for every distinct GPU type in the cluster using Algorithm 1. The output is an updated manifest file and memory and time predictions, which are then forwarded to the scheduler.

Algorithm 1 Controller

Input: Job manifest, GPU types (i: 1..n) in the cluster
Output: Job manifest with assigned node and allocated GPU memory

1: $X = extractCharacteristics(job\,manifest)$
2: **do in parallel** $\forall\, GPU\ Types$
   $P_{Mem,Time}(X) = Predictors(X)$
   end
3: $Scheduler(X, (P_{Mem,Time}(X)))$

E. Scheduler

The main task of the scheduler is to determine which node will be assigned to a job. To do so, it starts by examining the nodes and excluding those that will not be able to fit the job, i.e., nodes with GPU memory less than the job memory requirement. Algorithm 2 shows the steps that are involved in estimating the needed memory for this operation. Then the scheduler picks the GPU node with the shortest queuing delay of all candidate GPU nodes by evaluating Equation 2, i.e., a GPU node that will yield the minimum JCT for the incoming

job.

$$Delay_Q = Remaining_{running} + Estimated_{pending} \quad (1)$$

$$JCT_{Estimated} = Delay_Q + Estimated_{incoming} \quad (2)$$

The equation calculates the estimated incoming job completion time on candidate GPU nodes by calculating the sum of two values for each candidate GPU: (1) The queuing time at the candidate GPU, which is the sum of the estimated JCTs of the queued jobs and the remaining time of the currently running jobs (i.e., estimated JCT - elapsed time since the start of the job); and (2) the estimated JCT of the incoming job on the candidate GPU. The scheduler finally updates/adjusts the user's job manifest by adding three values: (i) the assigned node label using the `nodeSelector` field; (ii) the volcano scheduler name using the `schedulerName` field; and (iii) the estimated memory demand using Volcano's resource name `volcano.sh/gpu-memory` field.

Finally, the scheduler submits the job to be run via the Kubernetes API server to be scheduled by the Volcano scheduler. The job will be monitored by the *monitor* when it is executed. The default Volcano scheduler schedules the jobs in FIFO order. However, after leveraging node selectors, our approach effectively results in virtually maintaining a per-node queue of tasks.

Algorithm 2 Scheduler

Input: Job manifest, job characteristics (X), time and memory predictions (P)

Output: Job manifest with assigned node and allocated GPU memory

do in parallel $\forall\, GPU\ Types$
  $Est_{Mem,Time}(X) = Estimator(X, P_{Mem,Time})$
end
for $i\ in\ GPU\ nodes$ **do**
  if $Est_{Mem}(i, X) \neq OOM$ **then**
    $Candidate\ GPUs \leftarrow (i, Est_{Mem})$
  end if
end for
do in parallel $\forall\, Candidate\ GPUs$
  $Delay_Q = getDelay_Q(Est_{Time}(X))$
end
$(Node_{assigned}, Est_{Mem}) \leftarrow Min(Delay_Q)$
update job manifest
submit the updated manifest and monitor job

The memory and time predictions are only computed for every different/distinct GPU type in the cluster (i.e., in a threaded approach/in parallel), and not for every GPU node in the cluster. The calculation of the queuing delay of all candidate GPU nodes is a simple addition operation and a few local lookups to get the time estimates of currently running/queued jobs which are maintained in lists.

F. Monitor

The monitor collects job and GPU statistics by using `Kubectl logs`, `kubectl describe`, and `nvidia-smi` for future analysis and for fine-tuning/retraining the prediction models. Upon the completion of a job, the monitor collects the data elements of the predictors dataset, and inserts them for later fine-tuning/retraining (especially when new unseen jobs arrive) and analysis. The monitor includes a per job monitoring task, which is lightweight as it only collects the logs upon completion of the job.

G. Interference

In our motivational study (Section III), we observed that as the number of collocated jobs increase, the average job completion time increases as well. Moreover, the increase in the JCT is approximately linear with the increase in the number of collocated jobs, and the number of collocated jobs is also limited by the total amount of available GPU memory. To accommodate this in our design, we use a threshold on the number of jobs that can be collocated on a GPU. This number depends on the type of the GPU and available memory. Our tests show that although simple, the approach is able to deliver good performance when collocating multiple jobs on a GPU.

V. Evaluation

A. Testbed

We run our experiments on a self-managed on-premises Kubernetes cluster of 5 nodes. Our focus is on heterogeneity of the resources. Two nodes have 32-core AMD Opteron Processor, 64 GB RAM, and a NVIDIA GeForce RTX 2070s-8GB GPU. Among the remaining three nodes, two have 64-core Intel Xeon Processor, 192 GB RAM, and an NVIDIA GeForce RTX 3070-8GB GPU. The last node has a 20-core Intel Core i9 Processor, 32 GB RAM, and an NVIDIA GeForce RTX 3090-24GB GPU. Table I shows the characteristics of the GPUs used in our setup. All the nodes are connected with a 10 Gbps Ethernet connection. In addition to the local setup, we also perform experiments using AWS EKS clusters, with nodes equipped with NVIDIA Tesla K80-12GB (p2.xlarge) and Tesla V100-SXM2-16GB (p3.2xlarge) GPUs.

In all our experiments, we deploy Kubernetes v1.18.18. We create our on-premises cluster using v1.18.18 of kubeadm, kubelet, and kubectl. We use Docker v19.03.13 as the container runtime. We deploy Volcano [9] scheduler v1.3.0 and device plugin v1.0.0 for GPU memory sharing capabilities. We implement SchedTUNEas a proof-of-concept using Bash where we employ Kubectl. We implement the predictors in Python, and require pandas [49], scikit-learn [38], and Joblib [25]. Although we demonstrate/showcase the prediction models using Kubernetes, we believe that they can be applied to other systems such as HPC and GPUaaS systems by integrating GPU heterogeneity-aware intelligence into local schedulers, like SLURM.

B. Workload

Our workload is made up of 360 (312 inference jobs, 48 training jobs) different DL jobs with convolutional neural network models ResNets [16], DenseNets [19], and VGGs [43].

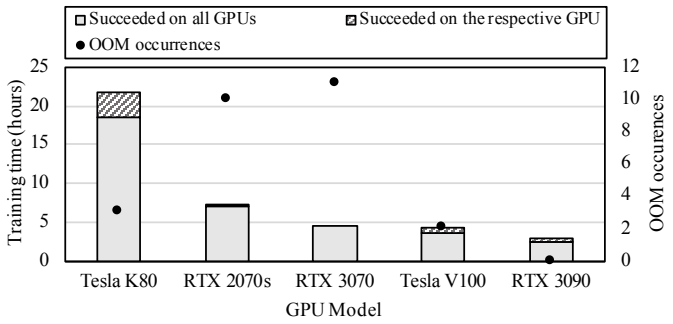

Fig. 11. Baseline of the total training time of training jobs and OOM occurrences on each of the studied GPUs.

TABLE II
REGRESSOR MODEL ACCURACY FOR GPU MEMORY PREDICTION, MEASURED USING ROOT MEAN SQUARED ERROR (RMSE), ROOT MEAN SQUARE LOG ERROR (RMSLE), AND AVERAGE ERROR (%).

Model	RMSE	RMSLE	Avg ERR (%)
Random Forest Regressor	118.7	0.02	1.56
Gradient Boosting Regressor	212.3	0.03	2.56
Voting Regressor	157.3	0.02	1.90
XGBoost for Regressor	232.6	0.03	2.75

TABLE III
THE IMPACT OF USING DIFFERENT GPU CHARACTERISTICS: (1) CUDA CORES, (2) TENSOR CORES, (3) MEMORY BANDWIDTH (GB/S), (4) SM COUNT, AND (5) MEMORY CLOCK SPEED (MHZ) ON THE RFR PREDICTION MODEL ACCURACY FOR TRAINING MEMORY USAGE.

GPU Char.	RMSE	RMSLE	Avg ERR (%)	Deviation (MB)
(1)	119.9	0.02	1.60	[-818.4, 429.3]
(2)	154.5	0.02	2.11	[-636.6, 683.3]
(3)	122.5	0.02	1.56	[-894.4, 399.2]
(4)	125.4	0.02	1.61	[-952.0, 340.6]
(5)	158.6	0.03	2.20	[-939.0, 726.4]
(1,3)	118.7	0.02	1.57	[-763.7, 322.9]
(1,3,4)	110.6	0.02	1.50	[-624.7, 356.4]
(2,3)	129.6	0.02	1.66	[-646.3, 546.3]
(2,5)	140.0	0.02	1.95	[-786.5, 550.2]
(1,2,3,4,5)	118.1	0.02	1.50	[-656.4, 347.1]

TABLE IV
ESTIMATORS ACCURACY USING A SMALLER WORKLOAD (160 JOBS - SEE FIGURE 12) FOR FAIR COMPARISON TO MATCH WITH THE LIMITED SET OF HORUS [54] ESTIMATES.

Estimator	RMSE	RMSLE	Max ERR (%)	Min ERR (%)	Avg ERR (%)
Horus	8908.5	0.62	269.32	0.13	77.56
Torchinfo	6826.2	0.51	224.30	0.39	56.04
SCHEDTUNE	196.1	0.04	18.38	0.05	3.83

All the training jobs train on the Cifar10 [26] dataset and use SGD [42] as optimizer. The training dataset image size does not effect the GPU memory utilization. This is because all images are resized (i.e., preprocessed on the CPU) when the DL model starts execution. In our predictions, the input size is the resized size. For training jobs, we vary the batch sizes from 8 to 64. We run the training job for one epoch only. Here, an epoch is a single pass over the entire dataset and is a representative of the training cycles performance on the GPU. Our workloads utilize the DL framework provided by PyTorch:1.8.1 with CUDA 11.1 [10] and cuDNN 8 [8].

C. Baseline

For our baseline numbers, we run our workload on the standard Kubernetes cluster. Figure 11 shows the total training time of sequentially running training jobs on each of the GPUs we consider in this study. We see a significant performance variation across the GPUs. The figure also shows the number of occurrences of OOM errors on each GPU. We observe that on the RTX 3090 GPU, no job fails due to OOM. This is because the memory capacity of the RTX 3090 GPU is significantly high compared to the memory requirements of the jobs in the workload. However, some jobs fail on other devices as seen. This highlights the importance of considering the GPU device characteristics for avoiding OOM errors.

D. Prediction Models Accuracy

In our next set of experiments, we determine the accuracy of the prediction models by measuring their Root Mean Square Error (RMSE), Root Mean Square Log Error (RMSLE), and average error (%). As shown in Table II, the Random Forest Regressor achieves the lowest RMSE, RMSLE, and average error (%) across all other regressors when training on our GPU memory prediction dataset. This shows that the Random Forest Regressor is the best suited amongst all the others, and we thus choose it for use in SCHEDTUNE.

Next, we measure the impact/relativeness of the considered GPU characteristics (alone and in combinations with others). Table III shows the RMSE, RMSLE, average error (%), and memory deviation range in MB for the studied cases as shown. We observe that considering a combination of CUDA cores, Memory Bandwidth (GB/s), and SM count yields the lowest RMSE and RMSLE. Interestingly, considering all five of the characteristics together yields the second lowest error rate.

Next, we compare our GPU memory predictions with the actual recorded GPU memory requirement, as well as with the predictions of two state-of-the-art systems Horus [54] and Torchinfo [46]. For fairness, in this evaluation, we only compare the GPU memory predictions for the values which we could obtain from Horus given system-level differences. We vary the batch sizes from 8 to 64. Figure 12 shows our predictions for the RTX 2070s GPU. We see that, compared to the existing approaches, our predictions are the closest to the actual memory requirements. We also measure the RMSE, RMSLE, maximum, minimum, and average error (%) of Horus and Torchinfo predictions and compare to SCHEDTUNE. Table IV shows the results. Finally, we also consider a close-source tool, DNNMem [12], which reports maximum, minimum, and average prediction errors of 23%, 7.5%, and 14.4%, respectively. Thus, SCHEDTUNE is able to perform better than DNNMem by considering GPU characteristics.

Next, we measure the RMSE, RMSLE, maximum, minimum, and average error (%) of SCHEDTUNE predictors as shown in Table V. We deploy the predictors that yield the highest prediction accuracy. For example, to predict the

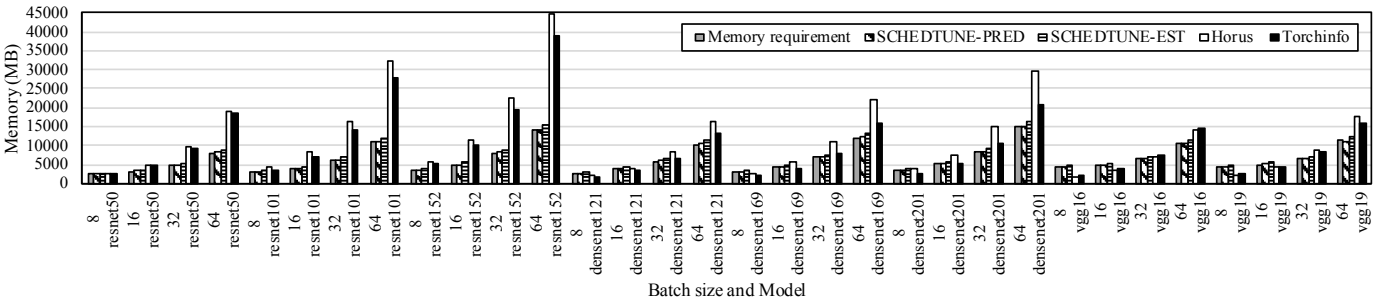

Fig. 12. Actual vs. predicted memory requirement for various DL models on Nvidia GeForce RTX 2070s GPU. We show the predicted and estimated (after adding the error (%)) memory values of SCHEDTUNE.

TABLE V
ACCURACY OF SCHEDTUNE WORKLOAD PREDICTION. M REFERS TO MEMORY AND T REFERS TO TIME. THE GPU CHARACTERISTICS (1,3,4) ARE USED FOR TRAINING MEMORY PREDICTION AND (1,2,3,4,5) ARE USED FOR THE THREE OTHER PREDICTIONS.

Predictor	RMSE	RMSLE	Max ERR (%)	Min ERR (%)	Avg ERR (%)
Train-M	110.6	0.02	8.98	0.01	1.50
Train-T	23.6	0.03	21.85	0.00	2.50
Infer-M	9.8	0.00	5.42	0.00	0.37
Infer-T	46.5	0.05	24.36	0.00	3.35

GPU memory requirement of DL training jobs, we use the combination (1,3,4) as defined in Table III. Overall, we see that SCHEDTUNE is accurately able to predict resource requirements for the heterogeneous workloads and heterogeneous GPUs.

E. GPU Memory Utilization

In our next test, we determine GPU memory utilization under SCHEDTUNE. For this experiment, we submit the jobs sequentially. Figure 13 shows the overall cluster-level GPU memory utilization of SCHEDTUNE compared to Kube-scheduler (the default Kubernetes scheduler). The total cluster-level GPU memory is 56178 MB. The GPU memory utilization of the Kubernetes scheduler is significantly low. This is because GPU sharing is not used. Moreover, on average six jobs exit with OOM error. This is because the Kube-scheduler does not differentiate between the GPU devices and does not consider DL model and GPU characteristics. In contrast, SCHEDTUNE is able to avoid these errors and provide successful job completion.

Figure 14 shows the cluster-level GPU utilization variation over time for SCHEDTUNE and Kube-scheduler. SCHEDTUNE achieves 81% (calculated as the ratio of average GPU memory utilization by SCHEDTUNE to that of Kube-scheduler) higher GPU memory utilization over Kube-scheduler by learning job requirements and better matching the jobs to appropriate GPUs.

F. Workload Makespan

Next, we study the workload makespan under SCHEDTUNE and the Kube-scheduler. Figure 15 shows the result. We observed that SCHEDTUNE took more time to choose resources; on average the overhead was observed to be 6% in our tests. The average overhead is skewed towards the shorter duration of inference jobs and the number of inference jobs in our workload, i.e., inference jobs account for 87% of our workload. However, by better matching resources with workload requirements, SCHEDTUNE is able to reduce the workload completion time by 17.5%. Additionally, all jobs complete successfully without OOM errors. Furthermore, the makespan under Kube-scheduler would be higher if jobs did not fail with OOM errors. Thus, SCHEDTUNE's advantage is ever more pronounced than just observed from this experiment. Recall, we calculate the makespan as the time duration between the start of the first job and the completion of the last job in the workload, whether it is a successful completion or failure due to OOM.

In summary, our results show that SCHEDTUNE is able to avoid OOM errors by performing smart heterogeneity-aware resource scheduling, and is able to perform at par with or better compared to the state-of-the-art GPU schedulers for DL workloads.

VI. RELATED WORK

A number of works have explored the use of GPUs for ML/DL workloads from a number of perspectives. In the following, we discuss works most relevant to ours.

a) GPU cluster management: Jeon et al. [24] characterized and analyzed deep neural networks (DNN) training workloads on a multi-tenant GPU cluster. They identified special cluster management requirements for DL workloads. Multiple works studied cluster scheduling for ML workloads [14], [22], [32], [39], [50], [51]. Unlike our study, all of these works consider homogeneous GPUs.

b) Resource scheduling for heterogeneous/hybrid clusters: Allox [31] introduces a scheduler that leverages the interchangeability of CPU-GPU resources at the application level. Allox does not take into consideration distributed jobs or GPU heterogeneity. Moreover, Allox only considers Nvidia K80 GPU that has a low performance compared to the state-of-the-art GPUs. Hu et al. [18] consider both load balancing and application-level container dependency awareness when scheduling on a heterogeneous CPU cluster. Specifically, they propose a scheduler that leverages container consolidation and heuristic bin packing. Chaudhary et al. [7] proposed user-level

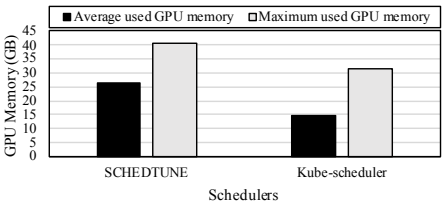

Fig. 13. The overall average and maximum cluster-level GPU memory utilization over the run time of the workload. The total cluster-level GPU memory capacity is 54.8 GB.

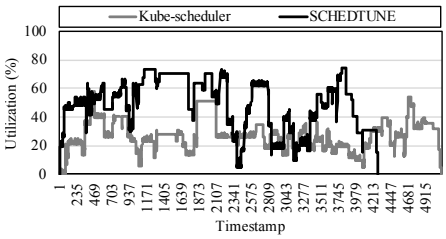

Fig. 14. The overall cluster-level GPU memory utilization.

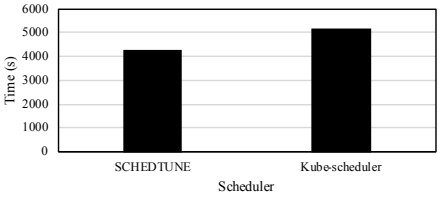

Fig. 15. The measured workload makespan for SCHEDTUNE vs. the default Kubernetes scheduler.

fairness in scheduling but did not consider GPU sharing like SCHEDTUNE.

c) Sharing of GPU resources: Recent works have introduced GPU sharing capabilities for Kubernetes clusters including KubeShare [53] and Kube-knots [45]. Major cloud providers have introduced GPU sharing capabilities such as Volcano [9], and GPU Sharing Scheduler Extender [3]. These solutions provide container-level GPU memory sharing of NVIDIA GPUs. SCHEDTUNE leverages and builds on such works to design application-attuned GPU resource scheduling.

d) Predicting GPU memory usage: Recent work has also proposed methods on predicting the memory demand of DL jobs. Gao et al. [12] proposed DNNMem that looked into instruction-level memory usage that can help provide insights into how much memory an application would use. Similarly, Yeung et al. [54] proposed a linear equation to estimate the memory demand and a regression model to predict GPU utilization. However, in contrast to SCHEDTUNE, these studies do not consider GPU heterogeneity and different GPU characteristics in their memory prediction models.

e) Leveraging GPU characteristics: A recent work [4] offers an experimental analysis of GPU execution time prediction using machine learning models. The paper analyzes different GPU characteristics to determine the most important characteristics for considering in GPU execution time prediction. This work is helpful to our approach, and we learn and leverage its findings. Additionally, the results from our GPU characteristics experiments align with the results presented in the paper. Lattuada et al. [30] proposed leveraging the computational power of GPUs, measured in single precision GFlops/s, as a metric representing GPU performance. They utilize a linear regressor to predict the training time of DL workloads but do not consider GPU memory prediction or GPU sharing like SCHEDTUNE.

VII. CONCLUSION

Container management systems are beginning to support heterogeneous GPU resources. However, existing solutions are unable to address the challenges of GPU under-utilization, high queuing delays, subsequent low throughput, and high makespan. Moreover, sharing GPUs across multiple applications via collocation, while needed, is not fully supported, especially because misconfiguration of DL jobs' GPU memory requirements can significantly reduce performance. In this paper, we address the above issues by designing SCHEDTUNE, an ML-based GPU heterogeneity-aware scheduler that utilizes DL jobs predictions for GPU memory requirement and job completion time. The predictions consider the characteristics of GPUs along with the characteristics of the DL model to produce GPU-specific estimates. The evaluation of SCHEDTUNE on representative workloads shows an 81% increase in GPU memory utilization, 100% prevention of OOM errors, and 17.5% reduction in makespan compared to the state-of-the-art Kubernetes scheduler.

ACKNOWLEDGMENT

We are thankful to the reviewers and our shepherd Ana Lucia Varbanescu for their valuable feedback. This work is sponsored in part by the NSF under the grants: CSR-2106634, CCF-1919113, OAC-2004751, and CSR-1838271, by Kuwait University, and by Guangdong Basic and Applied Basic Research Foundation No. 2021A1515110080.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in {*USENIX*} {*OSDI*}, 2016.

[2] "Container Service for Kubernetes," Alibaba Cloud, https://www.alibabacloud.com/product/kubernetes.

[3] "GPU Sharing Scheduler Extender," Alibaba Cloud, https://github.com/AliyunContainerService/gpushare-scheduler-extender.

[4] M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, "A comparison of gpu execution time prediction using machine learning and analytical modeling," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2016.

[5] "Launching applications using docker containers," Apache Hadoop 3.1.1, https://hadoop.apache.org/docs/r3.1.1/hadoop-yarn/hadoop-yarn-site/DockerContainers.html.

[6] "Amazon Elastic Kubernetes Service (EKS)," AWS, https://aws.amazon.com/eks.

[7] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *EuroSys*, 2020.

[8] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[9] "Volcano: A cloud native batch system," Cloud Native Computing Foundation, https://www.cncf.io/projects/volcano/.

[10] "Compute Unified Device Architecture," CUDA, https://developer.nvidia.com/cuda-zone.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[12] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating gpu memory consumption of deep learning models," in *ESEC/FSE*, 2020.

[13] "Google Kubernetes Engine (GKE)," Google Cloud, https://cloud.google.com/kubernetes-engine.

[14] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A {GPU} cluster manager for distributed deep learning," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 485–500.

[15] J. Han, L. Xu, M. M. Rafique, A. R. Butt, and S.-H. Lim, "A quantitative study of deep learning training on heterogeneous supercomputers," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[17] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[18] Y. Hu, C. T. A. M. de Laat, and Z. Zhao, "Multi-objective container deployment on heterogeneous clusters," in *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019, Larnaca, Cyprus, May 14-17, 2019*. IEEE, 2019, pp. 592–599. [Online]. Available: https://doi.org/10.1109/CCGRID.2019.00076

[19] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[20] "Cloud Container Engine (CCE)," HUAWEI CLOUD, https://www.huaweicloud.com/intl/en-us/product/cce.html.

[21] "Volcano device plugin for kubernetes," HUAWEI CLOUD, https://github.com/volcano-sh/devices.

[22] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park, "Elastic resource sharing for distributed deep learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 721–739. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/hwang

[23] "IBM Cloud Kubernetes Service," IBM Cloud, https://www.ibm.com/cloud/kubernetes-service.

[24] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 947–960.

[25] Joblib Development Team, "Joblib: running Python functions as pipeline jobs," https://joblib.readthedocs.io/.

[26] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[27] "Kubernetes," Kubernetes, https://kubernetes.io.

[28] "Schedule GPUs," Kubernetes, https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

[29] "What is Kubernetes?" Kubernetes, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes.

[30] M. Lattuada, E. Gianniti, D. Ardagna, and L. Zhang, "Performance prediction of deep learning applications training in gpu as a service systems," *Cluster Computing*, pp. 1–24, 2022.

[31] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, "Allox: compute allocation in hybrid clusters," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[32] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient {GPU} cluster scheduling," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 289–304.

[33] "Azure Kubernetes Service (AKS)," Microsoft Azure, https://azure.microsoft.com/en-us/services/kubernetes-service.

[34] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "{Heterogeneity-Aware} cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 481–498.

[35] "Nvidia system management interface (nvidia-smi)," NVIDIA, https://developer.nvidia.com/nvidia-system-management-interface.

[36] "Determine memory cuda context memory usage," Nvidia Forums, https://forums.developer.nvidia.com/t/determine-memory-cuda-context-memory-usage/67460/12.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[39] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[40] "Red Hat OpenShift," Red Hat, Inc., https://www.redhat.com/en/technologies/cloud-computing/openshift.

[41] "State of kubernetes security report," RedHat, https://www.redhat.com/rhdc/managed-files/cl-state-kubernetes-security-report-ebook-f29117-202106-en_0.pdf.

[42] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[44] "Apache Mesos," The Apache Software Foundation, http://mesos.apache.org/.

[45] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–13.

[46] Tyler Yep, "Torchinfo: Model summary in PyTorch," https://pypi.org/project/torchinfo/.

[47] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 353–362.

[48] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia, "Characterizing deep learning training workloads on alibaba-pai," in *2019 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2019, pp. 189–202.

[49] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[50] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 595–610.

[51] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on {GPU} clusters for deep learning," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 533–548.

[52] X. Xu, N. Zhang, M. Cui, M. He, and R. Surana, "Characterization and prediction of performance interference on mediated passthrough gpus for interference-aware scheduler," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[53] T.-A. Yeh, H.-H. Chen, and J. Chou, "Kubeshare: a framework to manage gpus as first-class and shared resources in container cloud," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 173–184.

[54] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[55] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1159–1170.